



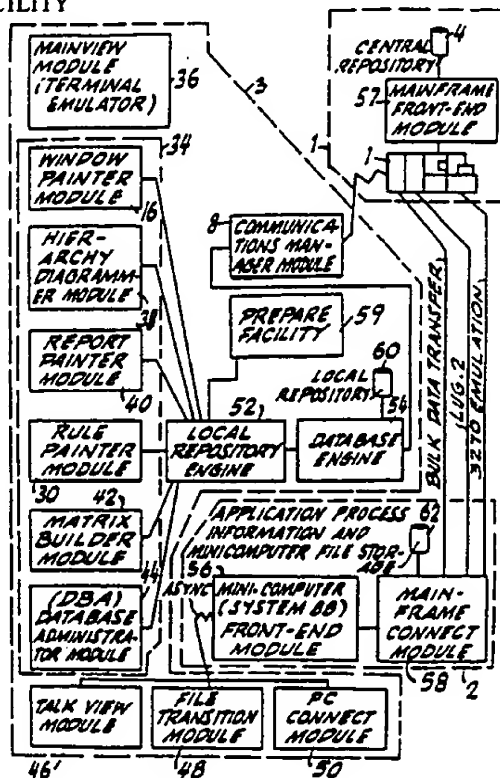
INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

| | | |
|---|----|--|
| (51) International Patent Classification ⁵ : | A1 | (11) International Publication Number: WO 91/08543 |
| G06F 15/40, 15/21, 9/45 | | (43) International Publication Date: 13 June 1991 (13.06.91) |
| <p>(21) International Application Number: PCT/US90/07013</p> <p>(22) International Filing Date: 30 November 1990 (30.11.90)</p> <p>(30) Priority data: 444,060 30 November 1989 (30.11.89) US</p> <p>(60) Parent Application or Grant</p> <p>(63) Related by Continuation US 444,060 (CIP) Filed on 30 November 1989 (30.11.89)</p> <p>(71) Applicant (for all designated States except US): SEER TECHNOLOGIES, INC. [US/US]; 5 Penn Plaza, New York, NY 10001 (US).</p> <p>(72) Inventors; and</p> <p>(75) Inventors/Applicants (for US only) : WADHWA, Vivek, K. [IN/US]; 474 Lincoln Drive, Paramus, NJ 07652 (US). ATAIE, Faraz [IR/US]; 492 11th Street #4D, Brooklyn, NY 11215 (US). AUBRUN, Vincent, P. [FR/US]; 315 West End Avenue Apt. 7A, New York, NY 10023 (US). ERLIKH, Leonide [US/US]; 2819 West 12th Street #16P., Brooklyn, NY 11224 (US). FISCHER, Michael [US/US]; 240 Park Avenue, Passaic, NJ 07055 (US). FOCHLER, Michael [DE/US]; 301 Cathedral Parkway Apt. #14G, New York, NY 10026 (US). HAYMAN,</p> | | <p>Craig, B. [GB/US]; 377 Rector Place #22E, New York, NY 10280 (US). HILDEBRAND, Daniel [US/US]; 38 Ranson Street Apt. 2B, Stamford, CT 06902 (US). HUGHES, James [US/US]; 36 Wilson Street, Hartsdale, NY 10530 (US). LAMBERT, Jeffrey, L. [US/US]; 8 Tremblay Road, East Brunswick, NJ 08816 (US). LEE, Douglas, E. [US/US]; 10 Lake Street Apt. 3D, White Plains, NY 10603 (US). LIM, Nicholas, R. [GB/GB]; 201 C Tufnell Park Rd., London N7 0PU (GB). MODI, Rajan, S. [IN/US]; 372 CPW 48, New York, NY 10025 (US). MOSEBACH, Richard, W. [US/US]; 100 Sixth Street, Hicksville, NY 11801 (US). MOSKOWITZ, Joel, M. [US/US]; 480 East 74th Street Apt. 3B, New York, NY 10021 (US). OLOWU, Tayo [GB/US]; 630 First Avenue Apt. 23B, New York, NY 10016 (US). POWER, Elaine, C. [GB/US]; 54 St. Marks Place, Apt. 2, New York, NY 10003 (US). SHING, Norman [US/US]; 710 New Hyde Park Road, New Hyde Park, NY 11040 (US).</p> <p>(74) Agents: BRAINARD, Charles, R. et al.; Kenyon & Kenyon, One Broadway, New York, NY 10004 (US).</p> <p>(81) Designated States: AT (European patent), AU, BE (European patent), CA, CH (European patent), DE (European patent), DK (European patent), ES (European patent), FR (European patent), GB (European patent), GR (European patent), HU, IT (European patent), JP, KR, LU (European patent), NL (European patent), SE (European patent), SU, US.</p> |
| | | Published With international search report. |

(54) Title: COMPUTER-AIDED SOFTWARE ENGINEERING FACILITY

(57) Abstract

A computer-aided software engineering facility (CASE) for providing a method for generating source code and executable computer programs in a language supported by various hardware entities in the multiprocessing system, using the entity-relationship model and the high level Rules Language models that are distributable across multiple hardware environment or platforms. An object oriented modeling system is complied with linked rules of the Rules Language (59) as well as other system components to quickly and efficiently design computer source code and executable computer modules that have a high degree of precision as stored in a Repository (52).



-2-

requires many man-hours. Experience in the programming language used to develop the program is necessary.

With the code written, the design team's next problem is to debug the program for syntax errors, and
5 then test the program to determine whether the application performs the desired function. Typically, the debugging and testing phase requires the programmer to evaluate a program at the code level.

The programmer's development tasks become more
10 challenging with the use of multiprocessing systems architecture.

Traditionally, there have been two basic hardware configurations employed in the design of computer systems for multiple users. In one
15 configuration all of a system's processing is performed by one large "mainframe" computer. Each user accesses that system through non-processing terminals.

A network system is the second traditional hardware configuration. A network is comprised of a
20 number of individual processing units that are interconnected to allow the sharing of data and software. There may be additional processors to maintain a group's centralized database and additional processors may be dedicated to the maintenance of the
25 system's operation. The use of a network of processors each dedicated to specific aspects of a computer system is the essence of multiprocessing.

The growth of the use of multiprocessing systems can be traced to the proliferation and
30 development of powerful "micro" or "personal" computers (PC's). Within certain constraints, a PC can perform many of the same processing tasks that mainframe or mini computers do. However, a PC can perform these tasks with substantial savings of instructions executed by the

-1-

COMPUTER-AIDED SOFTWARE ENGINEERING FACILITYField of Invention

The invention relates to multiprocessor computer systems and in particular to facilities that aid in the development of computer programs for those systems.

Background of the Invention

Computer-Aided Software Engineering (CASE) software or "facilities" assist computer programmers or system developers in the design, development and testing of a computer program.

Traditionally, the steps necessary to implement a computer program were performed manually. Design teams follow a number of discrete steps to create a useful application program from a nascent idea.

The analysis begins by manually developing a model in which a problem can be solved by computer. Design teams evaluate the needs of the prospective users and the properties that the computer system should possess to meet those needs.

In the technical design phase of development, developers begin to define how the application program will be built on a given system. They manually determine the procedural and data elements needed and how the data and procedures will be assembled to form the software solution. At this phase the two major tasks are data-modeling and process-modeling.

With the basic design of the program modeled, developers then begin the task of coding the program. Computer programs are generally written in high-level programming language such as BASIC, C, COBOL, FORTRAN or PL/I. The task of reducing the theoretical design of an application to working code is an arduous task that

-3-

computer, measured in millions of instructions per second, "MIPS". Moreover, unlike a mainframe system, a PC is dedicated to a single user. An efficient multiprocessing system encourages data sharing and the use of dedicated PC's for as many tasks as possible.

In multiprocessor systems, an application may be executed on more than one processor. When various parts of an application program are executed on separate processors, the application is "distributed."

10 Distributed processing can be executed either in a serial sequence or in parallel.

The simultaneous execution of a program on many processors is parallel processing. Sequential execution of a program across different hardware environments is serial or "cooperative" processing.

Multiprocessing capability brings new challenges to computer system designers. Whereas in a mainframe environment all parts of the program were programmed for a single environment, in multiprocessing systems designers can choose the particular environment where specific aspects of a program will run.

Although this freedom to distribute the program results in a highly efficient application, the programmers must now construct programs designed to execute in many hardware environments.

The task of programming multiprocessing systems involves difficult problems of swapping data across different, and incompatible environments. For example, a file containing data stored in a PL/I format is not readable by a program written in a different language such as C or COBOL. A programmer must design special software to handle the communication problems inherent in multiprocessing systems.

In addition, a programmer must code the

-4-

various distributed programs in the language supported by that environment. For example, if the PC processors support only programs written in C language, those parts of the program must be in C. Whereas the other parts of the program, such as those on the mainframe, may have to be written in another programming language.

Programming in a multiprocessor environment is very complex and time consuming. Staffing requirements for designing an application in a multiprocessor environment alone can make the task cost prohibitive. CASE facilities were created to alleviate some of the burden multiprocessing architecture placed on the programmer.

Traditionally, CASE facilities allow a user to input a high-level logical design of a program which is then translated into code in a particular computer language. However, the CASE facilities currently available do not give the system developer a completely integrated system for the design, implementation and maintenance of a software application. The current facilities do not support the development of a program in a centralized location and where translated code is created and distributed to various environments. The current CASE tools also do not provide a method for re-using parts of a previously developed application that may be usable in an application under development. The current CASE facilities do not provide adequate testing and debugging tools.

One of the dilemmas in designing a CASE tool is that the output one is required to provide the user for the planning phases of the project is different from the output required later on, during implementation. Early CASE tools tended to emphasize support for one of these two categories of activity, and scrimped on the

-5-

other.

With prior design-oriented CASE tools, the output of a designer's work with the tool was a picture. What a designer would end up with is documentation that could be used to actually create a system, but not the actual system itself. In addition, if during the implementation phase, a user discovered a reason to change any aspect of the program, the designer had to go back and manually identify the change and update the documentation -- the documentation was useless unless it was compulsively updated. The actual implementation of the system, as well as any changes, were done using traditional methods. The only advantage of using a design-oriented CASE tool was that it produced friendly, complete documentation that really helped in design at the early stages.

The second family of CASE tools were implementation-oriented. They allowed a designer to draw a diagram of an entire model, but not of a piece of a module. Essentially, the designer could get a picture of whatever model the designer was working on. When the designer unloaded the model to the mainframe repository, the model replaced whatever model was already stored there. The actual documentation of these systems was primitive and inflexible.

Summary of the Invention

The present invention is a CASE facility providing a method and apparatus for designing, implementing and maintaining software that runs in multiple hardware environments -- a distribution that is transparent to the end-user.

One aspect of the invention is a method of modeling the program structure and the data used by the

-6-

program through an entity-relationship modeling technique.

Another aspect is a relational database, where data is stored according to the entity-relationship
5 model.

Another aspect is the high-level Rules Language in which the logic of a program can be specified in a highly modular form, promoting ease of re-usability.

10 The invention provides the capability for generating source-code for the program in a language supported by various hardware entities in the multiprocessing system, using the entity-relationship model and the high-level Rules Language modules.

15 To generate the program's source-code the invention provides basic program elements to execute the program in a multiprocessing environment. These program components and low-level routines are combined with the Rules Language modules and data type entities to create
20 the program.

The invention further provides facilities to distribute, assemble and compile the program developed using the CASE facility of the present invention, as well as test and modify the program.

25

Brief Description of the Drawings

FIG. 1 is an illustration of a three-tiered computer system.

FIG. 2 is a diagrammatic representation of the
30 program elements according to the present invention.

FIG. 2A is a diagrammatic representation of sample developmental workbench modules placed in a PC hardware environment.

FIG. 3 is a diagrammatic representation of a

-7-

sample entity-relationship according to the present invention.

FIG. 4 is a diagrammatic representation of a sample entity decomposition according to the present invention.

FIGS. 5 and 6 are diagrammatic representations of the entity-relationship model for application programs according to the present invention.

FIG. 7 is a diagrammatic representation of the Process Flow for a sample program using entity-relationship modeling techniques according to the presented invention.

Detailed Description of the Invention

15 A. Hardware

A typical hardware configuration for a computer system using the CASE facility of the present invention is shown in FIG. 1. The figure describes a "three tiered" computer system, named for the three distinct types of processors networked: the mainframe computer 1, as for example, an IBM 3090, a mini or supermini computer 2, as for example an IBM S/88 or Stratus, and a plurality of micro-computer workstations 3, as for example IBM PC workstations. A system using the CASE facility of the present invention is not limited to only this hardware configuration. For example, a similar CASE facility can be constructed to develop code distributed in a two-tiered environment, such as a system employing only a mainframe and PC workstations or a two-tiered environment comprising a mainframe computer and a minicomputer with both accessible by non-intelligent terminals. The CASE facility can be tailored to fit any hardware configuration. The basic principles of code generation,

-8-

the maintenance of a centralized database and the use of a rules-based language used in conjunction with other development tools presented by this invention are the same regardless of a particular hardware configuration.

5 Flexibility in hardware configuration is one advantage of the CASE facility according to the present invention over those CASE facilities currently available. The CASE facility of the present invention allows flexible architectural designs by creating a
10 centralized database called a Repository where the design and the logic of the program module is stored, and translated code is distributed from the repository automatically to the different hardware environments.

Typical processors preferred for each of the
15 tiers in the exemplary three-tiered hardware configuration are the IBM mainframe sold under the trademark "Model 3090", running the IBM MVS/XA operating system; the Stratus mini-computer sold under the trademark "Model XA2000", running the Stratus Computer,
20 Inc. VOS operating system, and the IBM micro-computer sold under the trademark "PS2", executing the Microsoft Corporation operating system sold under the trademark "MS-DOS." (For further information on these processors or their operating systems, the reader is referred to
25 the following publications that are hereby expressly incorporated by reference: "IBM System/370 Bibliography", document number 6024448; and "Introduction to VOS", by Stratus Computer, Inc., document number R0001.)

30

B. CASE Facility Elements

The CASE facility of the present invention comprises a number of programmed elements, as depicted in FIG. 2: a Repository 4; a Rules Language 6; a

-9-

Communications Manager 8; a Testing/Debugging Facility 10; a PC User-Interface Facility 2, including a Rules Painter 30; a Work Station Utility comprising a Work Station Manager 14, a Window Painter 16, a Workstation Converse 18; a Documentation Facility 20; and a Systems Generator 22 including a Code Generator 20, a Data Access Generator 16, and Systems Assembler 28. Although the programs created by the CASE facility are centrally-stored, the CASE system program elements may be located in any of the hardware environments used in a configuration.

The Repository 4 is a central database used to store all information about all of the application programs created with the CASE facility of the present invention. The Repository 4 could exist in any hardware environment supporting a standard relational database. For example on the three-tiered environment shown in FIG. 1, the Repository 4 could exist on the mainframe computer 1, using the IBM relational database management system DB2.

For purposes of a preferred embodiment of this invention, it is preferred that Central Repository be located in the mainframe environment and that the IBM relational database sold under the trademark "DB2" be employed. (For background information on DB2, the reader is referred to the following IBM publications which are hereby expressly incorporated by reference: "IBM DATABASE 2 Introduction to SQL" (document number GC26-4082); "IBM DATABASE 2 Reference" (document number SC26-4078); "OS/VS2 TSO Command Language Reference" (document number GC28-0646); "TSO Extensions Command Language Reference" (document number SC28-1307); "Interactive System Productivity Facility/Program Development Facility for MBS: Program Reference"

-10-

(document number SC34-2089); "Interactive System Productivity Facility/ Program Development Facility of MVS: Dialog Management Services" (document number SC34-2137) and "DB2 Application Programming Guide for
5 TSO and Batch Users.")

The information stored in the Repository 4 includes models provided by the present invention to form the basis for an application program and high-level logic modules defined by the present invention for use
10 in the generation of an executable program, as will appear. For each program, a data-model and a process-model are developed through the use of an entity-relationship modeling system and stored in the Repository 4. The high-level logic modules stored in
15 the Repository 4 are written in a Rules Language defined by the present invention as described below. The information is environment independent and is structured to provide a high degree of re-use.

Programmers input information to the
20 Repository 4, using a standard database language supported by the hardware. For example where the Repository 4 is constructed from an IBM DB2 database, a programmer would use the DB2 Structured Query Language, SQL, to model the application. In another example
25 described below, graphic interface modules could be provided to the user to input the information.

The Rules Language 6 is a high-level language that permits users to specify the logic of a program, independent of the hardware devices used by the system.
30 Program modules written in the Rules Language 6 are translated by the Code Generator 24 into computer code suitable for execution in an environment where the modules are to run. The Rules Language is described more fully below.

-11-

The Communications Manager 8 performs the run-time transfer of information between hardware environments. For example the Communications Manager 8 would use routers and protocols to handle data transfers
5 between a mainframe, a mini computer and PC Workstations.

The Testing Facility 10, comprising a Rule View Module enables programmers to step through and debug program code. The Rule View Module can create
10 test data as well as provide regression and stress testing upon an application. The Rule View Module is explained more fully below.

The PC Front-End 12 allows PC based graphic interface to be used for all CASE tool functions. The
15 Rules Painter 30 permits programmers to construct program modules by manipulating graphic representations of the Rules Language statements. The PC Front-End 13 eliminates the need for the programmer to know an operating system language, such as DOS, by offering all
20 PC functions as menus.

The Workstation Manager 14 aids in managing user interface in a PC environment. The Window Painter Module 16 is a tool that helps the developer to create user-interface screens for applications. The
25 Workstation Converse 9 manages the display and validation of screen information.

The Workstation Manager Module 14 works in combination with commercialized programs to design user-interface, such as, for example, Microsoft Windows.
30 All of the complexities of using a commercial design tool, like Microsoft Windows, are managed by the Workstation Manager 14.

In designing an application or portions thereof for execution on an "IBM PC," it is preferred

-12-

that the Microsoft Corporation's operating system, sold under the trademark "MICROSOFT WINDOWS," be used. (For more background information on MICROSOFT WINDOWS, the reader is referred to the following Microsoft

5 Corporation publications which are hereby incorporated by reference: "Microsoft Windows Programmer's Utility Guide"; "Microsoft Windows Application Style Guide"; Microsoft Windows Programmer's Reference"; and "Microsoft Windows Quick Reference.")

10 The Documentation Facility 20 generates all technical documentation for a program under development. Documentation includes functional decomposition of the system and hierarchy listings of the Rule Language modules.

15 The System Generator 22 includes the Code Generator 12, the System Assembler 28, and the Data Access Generator 26. The Code Generator 24 translates the Rules Language modules into code in an appropriate programming language. The System Assembler 28 brings
20 the various coded program elements together to form an application program. The Data Access Generator 26 allows the program to access data across hardware environments. Code generation and program execution is discussed more fully below.

25 The PC-front-end 12, the Workstation Manager 14 and Communication Manager 8, together form a set of Development Workbench Modules 34 that enable the user to design the data and process models of an application, and rules language modules, and combine them to create a
30 fully functioning application program.

On the development platform in the exemplary configuration described above, access to all the environments is through the PC Front End 5. There developers can code their applications using the Rules

-13-

Language and the CASE facility will automatically generate native code for each environment, as appropriate.

However, for the last phases of development
5 for compiling and testing the applications, its necessary to switch to a front-end module for the appropriate execution environment of each one. With respect to those application development activities, each environment is accessed via an environment-specific
10 front end. The environment specific activity is described more fully below.

The Software Distribution System 32 automates and controls migration of an application. The system manages the release of software to targeted computers.
15 The Software Distribution System 32 solves the problem of synchronizing distribution of software located, for example, on hundreds of personal computers. For a more detailed description of a Software Distribution System see International Application No. PCT/US90/_____,
20 entitled "Software Distribution System;" filed on even date herewith in the name of Norman Shing et al., which is hereby expressly incorporated by reference.

C. Entity-Relationship Modeling

25 1. Entities and Relationships

A feature of the CASE facility of the present invention is the systems model. To design any application using the CASE facility, a developer must decompose the application into specified logical parts,
30 and assemble them into a program. The data used by a program are stored in Repository 4, FIG. 2, according to an entity-relationship model of the present invention. The different parts of an application are expressed as entities and are linked by relationships.

-14-

Defined generally, an entity is something real or abstract about which information is recorded. The information is organized into a set of characteristics, known as attributes. For example, collected information
5 about employees of a company could be placed in an entity type called Employee. The attributes for that entity could be a name, social security number, home address, age, birth date, department, etc. An entity called Organization would include attributes such as
10 organization name, address, type of organization (such as partnership or corporation), etc. This data is stored in a file in the Repository 4 (see FIG. 2).

An association between entities is known as a relationship. For example in FIG. 3 the entity,
15 Organization 64, is now linked to the entity, Employee 58, by the relationship, Employs 66. Relationships are also defined by attributes.

2. Functional Design

The functional design phase begins the
20 modeling task -- there are two tasks to perform: data-modeling and process-modeling.

Data-modeling is the method of creating an entity-relationship model for the real-world data to be used by an application program. The method involves
25 identifying an entity, such as a corporation, and decomposing that entity into subentities and attributes. In FIG. 4, the example entity, Corporation 70, is made up of the sub-entities: Product 72, Staff 74, and Customer 76. Customer consists of the attributes: Name
30 78, Address 80, and the Entity Order 82. The Order entity 82 is composed of the attributes: Number 84, Date 86, and Status 88.

Once the programmer has modeled the real-world data elements, he or she links them using the

-15-

relationships such as the Employs Relationship 3 illustrated in FIG. 3.

In model format, the data are stored in the Repository 4, (See FIG. 2). Using the entity-
5 relationship modeling technique, data for any application can be stored and readily re-used in subsequent applications.

Process-modeling is the method of constructing a model of an application program, using an entity-
10 relationship model. The CASE facility of the present invention requires that all applications be first reduced to an entity-relationship model, before a program's logic is specified. The entity-relationship model is not the actual program. It is separate from
15 the program modules written in the Rules Language. However, the CASE facility uses the entity-relationship model to link the various program modules together as well as to construct and distribute the program through a multiprocessor system. Entities store information on
20 the flow of a program, the data structures used, the user interface, the environments used by the modules and the multiprocessing needs of the program.

In addition the entity-relationship model provides a representation of the high-level design of
25 the programs from the entities. The CASE facility can produce technical documentation from the process model. For a given application, the entity-relationship model implicitly represents an amalgamation of a high-level structure-chart and a detailed description of the logic
30 and data requirements necessary to make a program run.

The process-modeling method allows users of the CASE facility to maximize efficiency by re-using program elements. The program resulting from the entity-relationship model is highly modular. Because

-16-

all programs developed with the CASE facility presented by this invention use the same entity types and relationships it is likely that many of the program elements can be re-used.

5

D. Application Model Entities and Relationships

As an example of the present invention, programs built using the CASE facility are broken into
10 ten entity-types and eleven relationships as shown in FIGS. 5 and 6. For each entity and relationship that defines an application a list of attributes and information is kept.

1. Entities

15 The Function Entity 90, FIG. 5, is a listing of all the application programs currently on the system. A Function 90 is defined by the following attributes: Function Name, Test Description and Application Identification.

20 A Process 92, FIG. 5, is a logical subdivision of a Function 90. A Function 90 typically is decomposed into two or more Processes 92. Processes 92 can be decomposed into lower level Processes 90 (sub-processes). When a Process 90 is related to another
25 Process 92, the relationship is always hierarchical. There are three Process types: Root, Leaf and Node. Such relationships define the decomposition of one subsystem into others. The Process Entity 92 is defined by the attributes: Process Name, Description, Menu
30 Description, Sub-Process Menu-Type and Sequence Number.

At runtime, a process 92 runs either as a foreground or background process. A Foreground Process is one that executes interactive communication with the end-user. For example, a Foreground Process may
35 comprise the graphics functions, resident on a PC

-17-

workstation environment 3, FIG. 1. This is the typical on-line/realtime process. In addition, the foreground processes can communicate synchronously with modules on other environments, and they can receive unsolicited data, asynchronously.

A Background Process, once started, will process its input, then stop, or wait for further input. This may be a batch process for example running on the mainframe, that receives input from a file, or it may be a continuous process that runs on-line, for example due that reads from a queue.

Rules are the procedural specifications of the logic of a Process. This logic is specified in a high-level language called the Rules Language. The Rules Language is based on the principles of structured design and programming. The syntax of the language along with restrictions built into the architecture ensure a highly modular and concise system specification process. The Rules Language is described in detail below.

Using Rules Language statements, the CASE facility generates source code for the various environments. The CASE facility of the present invention generates all code necessary to perform inter-system communication (for example, when a rule in one environment calls a rule in another), inter-process communication (for example, between different processes or regions on the same machine), program-to-program linkage, and user interface.

In the entity-relationship model only information about the Rule is stored in a Rule entity 94 -- not the Rule language statements. The Rule Entity 94, FIG. 5, FIG. 6, is defined by its attributes: Rule Name, Rule Description, Execution Environment; and Mode of Execution (for example, synchronous or asynchronous).

-18-

As with processes there are three categories of Rules: Root, Frontier and Node Rules. A Root Rule is invoked by a Process 92. Root Rules do not have user I/O capability. A Node Rule is any rule that is not a
5 Root or Frontier Rule.

Frontier Rules lie on the boundary of a new computing environment. Those Rules are grouped in a special category because Communication Manager 8, FIG. 2, must execute all Frontier Rules and their
10 input/output data has to be converted to accommodate changes in environment.

The Rule Relationships are modeled using Rule Entities 94 and the Uses Relationship 108.

Components are modules of code written in any
15 third generation programming language known to the CASE development system such as C, PL/I or COBOL. Components are used to perform functions not handled by the Rules Language such as calculations, database access, and calls to operating systems. The CASE facility assumes a
20 "black box" structure for Components. A black box has fixed inputs and outputs. Given a specific input, there is always a predetermined output. The same analogy applies to Components. Components have explicitly defined inputs and outputs.

25 The Component Entity 96, FIG. 5, FIG. 6 distinct from the component module described above contains the following attributes: Component Name; Component Description; Language Name (programming language that source code of the component is written
30 in); and Execution Environment.

Windows define user interface. They specify what data are to be accepted from a user, how it is to be displayed, and how it is to be accepted. The Window Entity 98 is used to store all information about the

-19-

user interface. The window information is used by the Code Generator Module 24, FIG. 2, and Workstation Converse Module 18, FIG 2.

A Window Entity 98, FIG. 5 in the entity-
5 relationship model has the following attributes: Window Name, Description. The Window Name contains the name of a Panel File also stored in the Repository which contains the information necessary to produce graphic interface using a graphic interface program such as
10 Microsoft Windows.

The View Entity 100, FIG. 5, FIG. 6, is a convenient grouping mechanism for storing data type variables that are used by the Rules Language. The View 100 is a hierarchical set of named scalar or aggregate
15 values.

Views 100 are is used in three different ways throughout a typical ystem. A File View 100A represents a template of the data constructs saved in a data file. The data constructs used to describe the input and
20 output to Rules and Components are Modular Views 100. The data structures used to handle user interface are called Window Views 100.

Field Entities 102, FIG. 5, FIG. 6, are the basic data elements that comprise the variables used in
25 a program. The Field Entity 102 stores all information about data elements, independently of the environment in which the data element is used. The Attributes of this entity define the format, editing specifications, report and screen headings, and any other generic information
30 about a particular data element.

A Set 116, FIG. 6 like a View 100, FIG. 5, FIG. 6, is a convenient grouping mechanism that is used to store related literals or constants. Value Entities 118, FIG. 6 are symbolic representations of literals or

-20-

constants. Value Entities 118 provide the ability or refer to specific data values by symbolic or English names. This eliminates the need to hard-code specific literal values in Rules.

- 5 Data are stored in files. In a entity-relationship model information concerning data files is stored in the File Entity 120, FIG. 6.

2. Relationships

- 10 The ten entities representing a program are linked together by one of eleven relationships.

 The Refines Relationship 104, FIG. 5, describes the decomposition of Functions 90 to Processes 92. Its attributes are: Function Name (first
15 participant); Process Name (second participant); and Sequence Number (for menu display). Processes also refine into further subprocesses.

 The Defines Relationship 106, FIG. 5, is used to describe the relation between an abstract Process 92
20 and the Rule Entities 94. Its attributes are: Process Name and Rule Name. Processes can also depend upon other processes.

 The Uses Relationship 108, FIG. 5, is used to describe the link between one executable module and
25 another: (e.g. Rule Uses Rule, Rule Uses Component and Component Uses Component). The attributes of a Uses Relationship are: Module Name and Sub-Module Name.

 The Converse Relationship 110, FIG. 5, describes the link between a Module and a Window Entity.
30 The attributes of the Converse 110 relation are: Rule Name (first participant) and Window Name (second participant). The function of the converse relationship is explained more fully below.

 The Relationship Owns 112, FIG. 5, connects

-21-

entities to the Views Entities 100 which describe their logical data interfaces. The Rule 94, Component 96, Window 98 and File 120, FIG. 6, entities may own View Entities 100. The attributes of the Owns relationship
5 are: Rule/Component/Window or File name; Entity type (either Rule/Component/Window/or file); View Name and View Usage (either In, Out or Inout for user interface I/O).

The Includes Relationship 114, FIG. 5 connects
10 data items together to form structures. View Entities 100 can include sub-Views 100 and Fields 102. The Attributes of the Includes Relationship 114 are: View Name (first participant higher-level view); View/Field Name (second participant); Entity Type (second
15 participant's entity type: either view or field); Sequence Number (used to order Sub-Views and Fields); Occurs Number of Times (used to create arrays in structures).

The Owns Set Relationship 124, FIG. 6, links
20 executable modules to the literal set values they refer to. A Rule may Own a Set by making a local declaration using the Rules Language DCL statement discussed below. The literal values will then automatically be included in the Rule Module when the code is generated. A
25 Component Entity 96 may also Own Sets 124. The Own Set Relationship Attributes are: Module Name; Module Entity Type (i.e. Rule or Component); Set Name.

The Contains Relationship 126, FIG. 6, link
literal data items to a common Set Entity 116. A Set
30 Entity 116 Contains, for example, 18 member Values in the Value Entity 118. The attributes of the Contains Relationship 126 are: Set Name; Value; Symbol Name; Sequence Number.

Finally, the CASE system has file

-22-

relationships: A File Entity 120, FIG. 6, is Keyed 132 by a Field Entity 122; File Entity 120 also Forwards 130 information to another File 120; and a File View Entity 100A Describes 128 a File Entity 120.

5

E. Process-Modeling Reduction Technique

All Applications developed with the CASE facility can be reduced to a set of pre-selected entities and relationships such as the above described
10 entities and relationships. The method begins by defining the Function Entity 90, FIG. 5. A Function is equivalent to any application program to perform any task or function. The Function Entity 1 contains a generalized description of the Application. A Function
15 is decomposed into processes. The processes represent general functions that the program must perform.

For example, one of the Process Entities 92 in a application to perform stock market trading could be entitled "Futures Trade Entry." A Process Entity 92,
20 FIG. 5, stores information describing the process. Processes can be decomposed into further processes. On each decomposition the information about the new process is stored in a Process Entity 92.

The Function Entity 90 and the Process
25 Entities 92 create a high-level overview of an application. During the modeling phase the CASE facility presented by this invention provides tools to aid development. The programmer can use CASE provided graphics to specify the Process 92 and Function 90
30 Entities and Refine Relationships 104. The user sees functions and processes listed on menus. The CASE facility also assists the analyst and user in process modeling by allowing this Function/Process decomposition to be prototyped interactively, as for example with an

-23-

analyst and a programmer. The analyst can enter his thoughts, generate menus reflecting the input, and solicit input (modifications) from the programmer as to whether this is in fact the way the user envisions the way a problem is to be solved.

Once the application problem has been reduced to a series of Function Entities 90 and Process Entities 92, technical design begins. In technical design the program coding specifications are described through the Rule 94, Component 96 and Window 98 Entities.

The important aspect of Technical Design is the specification of the Rules and Components, Windows, and Views that are necessary to run the program. FIG. 7 is an example of Process flow diagram using entity-relationship modeling standards. The Square boxes in the figure, 134, 136, 138, 140, 142, 144, represent Process and Function Entities. The capsule-shaped boxes, 148-182, represent Rules. The oval-shaped boxes, 184, 186, 192, 194, 196, represent Components. The circles, 188, 190, represent Windows. The process flow diagram is used to specify the information contained in the Rule 94, Component 96, Window 98 and View 100 Entities (see FIG. 5) as well as the Relationships between them.

Referring now to FIG. 2 there is illustrated Development Workbench Modules 34, used to generate the entity relationship model and rules language code. FIG. 2A illustrates an exemplary implementation of the Modules 34 in a three-tiered system, wherein the Development Workbench Modules 34 reside in the PC Environment 3. The Development Workbench Modules 34 comprise a series of graphic-based tools that help a user design and implement an application by providing a pre-selected set of entities and relationships and a

-24-

rules programming language. Everything that a user designs using the Development Workbench Modules 34 will be stored in a Local Repository 60 in the PC Environment 3. Eventually, the data in the Local Repository 60 are used to populate the Central Repository 4, which resides on the mainframe 1 environment. Everything that constitutes documentation or implementation of a system must be defined to the Central Repository 4, during the development cycle. Ultimately, the application is distributed, for example, to end-user PC workstation environments 3 (and other environments as appropriate) from the Central Repository 4.

However, use of the Development Workbench Modules 34 need not be connected to the Central Repository 4 at all times. For performance reasons, it is advantageous to avoid constantly querying the Central Repository 4 for information. Uploading from and downloading to the Central Repository 4 is done via the Communications Manager 8 that accesses to the Local Repository 60.

The Local Repository 60 is a storage device located in the PC Environment 3. A Database Engine Module 54 comprises software to store information and access it. A Local Repository Engine 52 provides the user access to the Development Workbench Modules 34 and enables the Development Workbench Modules 34 to access the Local Repository 60 via the Database Engine Module 54.

A Hierarchy Diagrammer Module 38 is provided in the Development Workbench Modules 34, to enable a user to specify the data and process modules that provide the logical foundation for the application program. The Hierarchy Diagrammer Module 38 could for example include an Entity-Relationship Diagrammer, which

-25-

could be used for data modeling -- i.e. specifying the entity relationship model for the real-world data that will be used by the application program. With the Entity Relationship Diagrammer, a user can graphically
5 identify entities and decompose the entities into sub-entities and defining attributes. For example, in FIG. 4, a Corporation 70 was decomposed into sub-entities Product 72, Staff 74 and Customer 76. The Entity Relationship Diagrammer provides modules to graphically
10 specify such entities and relationships.

The Development Workbench Modules 34 could also include a Matrix Builder Module 42 that enables a user to store, in matrix form, mappings between different types of entities. The matrix mappings are
15 typically used in the strategic planning phase of analysis to capture the relationships between data and process flow. The matrix created will also be saved in the Local 60 and Central Repositories 4.

Window Painter Module 16 enables a user to
20 create panels, which are the interface through which the end-users of an application interact. The Window Painter Module 16 panels determine the "look and feel" of an application.

Using the Window Painter Module 16, the user
25 can either create a new Window File for the related window entity or modify a previously created Window File that was stored in the Repository 4.

Using the example of a PC that is processing with the Microsoft Windows operating system, the Window
30 Painter Module 16 provides the software necessary for a user to create graphic interface in that environment.

For example, a user could access the Window Painter Module 16 from the Development Workbench, where a blank window could appear. A user could then select a

-26-

list of windows files previously defined in the Local Repository 60. Clicking on one of these Window names will bring up a panel upon which you can "paint" the objects that will be presented to the end-user of the user's application.

In addition to the panel screen, there are primarily two types of objects to manipulate: entity objects and local objects.

Entity objects are Field 102 and View 100 Entities that have been defined and reside in the Local Repository 60. A Views menu can provide a list of View Entities 100 that are attached to the Window that a user is painting, followed by the selectable Field Entities 102 and multiply-occurring sub-views that are in turn attached to them. A user can select any of these available objects, place them on the panel, and customize them.

Fields could appear as empty bracketed boxes with a screen label to the left. The value of the Field (if any) appears between the brackets at runtime. The screen label was defined when the Entity was created and is associated with the Field in the Repository 4. A user can edit the screen label as a text object on the panel. If the screen label field of the Field entity is blank, then the Field short name is used instead.

These objects could be moved about the panel by clicking on the object and holding down the mouse button while dragging the object to the desired location.

Local objects are associated with the panel but are not defined to the Local Repository 60. The user could select Local Objects from an Objects menu.

There are three types of Local Objects:

Control Objects, which can be selected by the

-27-

end user, with a mouse. Control Objects include pushbuttons, pushboxes, menu bar items, and pulldown items. Text Objects, which can be placed on the panel as labels or instructional messages.

5 By controlling placement of the screen objects, a user can build a window associated with a View Entity 100, and save that window specification for routine applications in the Local Repository 60 and in the Central Repository 4. The window files created in
10 this example will be used to create the user interface of an application program. The converse function described more fully below provides the software modules to link the user specified window file to the windows provided graphic interface routines.

15 The Rules Painter Module 30 provides the interface to allow users to create Rules Language Modules that are associated with the Rule Entities 94. The Rules Language Modules and Rule Entities are linked. Each Rule Entity contains information concerning the
20 functionaity of the rule and the name of a corresponding Rules Language Module. The corresponding Rules Language Module contains statements to execute the functionality of the linked Rule Entity. Typical statements comprising rules language are described more fully
25 below.

 The Report Painter Module 40 is used to design formats for end user reports in a manner similar to that of specifying windows for graphical interface using the Window Painter Module 16.

30 The Database Administration Module (DBA) 44 is a communications facility that enables a user to refresh data in the Local Repository 60 and upload changes to the Central Repository 4. The DBA Module 44 is also used to generate reports about the contents of the Local

-28-

Repository 60. This module uses the Local Repository Engine 52, the Database Engine 54 and the Communications Manager Module 8 in the PC Environment 3 to access data in the Local Repository 60, and send and receive data
5 from the Central Repository 4. Using the DBA Module 44, objects and attributes can be selected from the Repository 4 for alteration or deletion. The DBA Module 44 includes a system security component, requiring users to log in to the mainframe environment 1 and only
10 certain classes of users can have authorization to delete specific objects. The refresh feature enables users to download objects from the Repository 4.

The Development Workbench Modules 34 enables users to specify the data modelling necessary to specify
15 the entities and relationships, as well as to input the rules, components, views, field and window panels necessary to create the final executable program in later development steps. Using the Development Workbench Modules 34 these logic structures can be
20 stored in the Repository 4.

F. Constructing the Program

With the Entity-relationship model in place program construction can begin. Source code is
25 generated from the Rules Language Modules, Components, Windows, Files, Fields, Views, Values and Sets that are stored in the Repository. Rules, Components, Windows and Files are Program Modules distinct from the Rule Entities 94, Component Entities 96, Window Entities 98
30 and File Entities 120 linked in the entity-relationship model (see FIGS. 5 and 6). Those Entities store information about the actual program modules from which the application program is created.

At this stage of construction much of the work

-29-

in constructing the program modules is done. The data structures are defined by the View 100, Field 102, Set 116, and Value 118 Entities. The Code Generator Facility 24, FIG. 2, copies these entities into the
5 program modules that are related to them by the Rule 94 and Component 96 Entities (see FIG. 5). Component Modules do not need to be built by the programmer, as they are previously programmed subroutines provided by the CASE facility to perform basic mathematical
10 operations and operating system access. However, the CASE facility permits new components to be constructed and stored in the Repository 4, FIG. 2. The process of code generator is described more fully below.

15 1. Re-Usability Analysis

Even before the creation of the Rules Language Modules, using, as for example, the Rules Painter Module, the CASE facility speeds program development, because it provides a method to re-use previously
20 generated code. With entity-relationship modeling, much of the same logic from one application can be re-used in other applications. It is easier to re-use code that has already been developed and tested than to re-create it. Keeping this in mind, a re-use analysis should be
25 performed.

With the entity-relationship model stored in the Repository 4, FIG. 2, the Entities can be queried for usage by other Entities. For example, with the Repository 4 existing on a database supported by IBM
30 DB2, a Where Used query on the Field Entity 122, FIG. 5, returns a list of all the View Entities 100A using that Field Entity 122. The usage of the View Entities can be queried further to establish what Rules, Components or Windows use those View Entities. This method allows

-30-

programmers to discover previously coded Rules and Components. A Find Where query will search all entity attributes throughout the entire database for particular words or character strings.

5 In addition the CASE facility allows
programmers to define a keyword for all Rules,
Components and Field created. A Search command can be
performed to locate entities possessing a certain
keyword. A programmer can perform sophisticated
10 searches such as find all keywords starting with, for
example, "CU" and ending with "Pp".

Once a programmer has located a module that
may be re-usable, he or she can get further information
by browsing the description or other attributes
15 associated with the entity.

2. The Rules Language

The Rules Language is a high-level programming
language which supports all standard flow of control
20 constructs. What is unusual about the Rules Language is
that it does not require a means of describing elaborate
data structures, and it has data constructs which access
information in the entity/relationship modules. The
description of all the data structures used by a program
25 within the CASE facility is stored in the View and Set
Entities in the program's entity-relationship model.
All Rules used by a program share data structures. The
sharing of data structures provides strict coordination
between the data passed from one Rules Language Module
30 to another. This technique avoids one of the major
sources of program errors -- a mismatch between the data
passed between subroutines.

A Rule consists of zero or more declare
statements followed by zero or more executable

-31-

statements.

A data type declared must be one of the following type: Smallint, Integer, Char, Varchar, Decimal, Signed Picture or Like (an already declared Item). The declare statement is in the form:

DCL

declaration; [declaration; ...]

ENDDCL

where the declaration is:

10 identifier [(s)] [, identifier [(s)], ...]
 declare type or EXTERN identifier [,
 identifier, ...]
 SET

The data types are more fully explained in Appendix C
 15 attached hereto.

The Rules language supports three types of executable statements: 1) Assignment statements; 2) External flow of control statements; and 3) Internal flow of control statements:

| | | | |
|----|--|--|--|
| 20 | <u>Assignment</u> <u>Statements</u> | <u>External Flow</u> <u>of Control</u> <u>Statements</u> | <u>Internal Flow</u> <u>of Control</u> <u>Statements</u> |
| | MAP | USE | IF |
| 25 | OVERLAY | RETURN | CASEOF |
| | CLEAR | CONVERSE | DO |
| | | ASync | |

Assignment statements alter the data contained in and held by program variables. The MAP Statement
 30 places an expressed value in a variable location. The OVERLAY statement replaces the contents of a variable with specified data item. The CLEAR Statement replaces numeric fields with zero values and character fields with space values. The syntax of the assignment
 35 statements is:

MAP expression TO variable
 OVERLAY data item TO variable
 CLEAR variable

-32-

Four statements in the Rules Language handle the external flow of control between program modules. The USE statement enables one program module to invoke another Rule or Component. It is similar to a
5 subroutine CALL statement in other programming languages. The syntax is:

USE MODULE Component

USE RULE Rule [NEST]

The Next option indicates that all Windows invoking the
10 Rule will come in "pop up" mode; i.e., it will be displayed over the screen that was previously displayed.

The RETURN Statement transfers control back to the Rule that executed the USE statement:

15 RETURN.

The CONVERSE statement provides communication between PC-based rules and the user interface. A typical converse statement is:

CONVERSE WINDOW window

20 The converse statement invokes at runtime several modules provided by the present invention to physically execute the converse relationship between window and rule. In the example of a PC resident rule accessing a PC based window, the converse modules would
25 reside in the PC environment 3 and interface the operating system of that environment to provide graphic user interface.

The function of the converse modules is to manage all the screen input and output for a rule. The
30 modules send to the rule the user's input by populating a specified field. Prior to passing the end-users input to the rule conversing the window, the converse modules perform edit checking and validation of every input field specified by the attributes defined in the

-33-

repository. If for example there was a specified set of values related to a rule for a given field (e.g. if a color field could only equal "red" or "blue" any answer "green" would be an error).

5 In creating the user screen, it is also the function of the converse module to interface with the operating system to output the user interface screens specified by the window files.

 The Rules Language supports Asynchronous data
10 flow with the ASYNC statement. For example, in a system using a Stratus mini computer performing in parallel, unsolicited data allows a Stratus-based Rule to send data to a PC-based Rule. This is supported using the following statements:

```
15           ~     ASYNC     ATTACH  
              ~     ASYNC     DETACH  
              ~     ASYNC     REFRESH  
              ~     ASYNC     ROUTE
```

 The ASYNC ATTACH command will initiate the receiving of
20 unsolicited input as for example by a PC-based Rule sent by a Stratus-based Rule. The ASYNC DETACH command performs the exact reverse of ATTACH. It may only be used after an ATTACH command, at which point it will discontinue the receiving of unsolicited input. The
25 ASYNC ROUTE command switches data refreshing from one PC Rule to another. The ASYNC REFRESH statement will induce automatic updating of a specific field, so long as the Attach Statement has been used.

 The Rules language has three different
30 statements to order the flow of control within a Rule.

 An IF statement controls execution based on a specified condition:

```
              IF condition [ executable stmt ... ]  
              [ ELSE       [ executable stmt ... ] ]  
35           ENDIF
```

-34-

The CASEOF statement selects one of several alternative execution paths based on the value of a variable. The syntax is

```

CASEOF variable
5      CASE constant [ constant ... ] [ executable
      stmt
      ... ]
      [ CASE constant [ constant ... ] [
executable
10     stmt ... ]
      [ CASE OTHER
      ENDCASE

```

A DO statement controls the execution of repetitive loops. The syntax is

```

15      DO [ FROM data item ] [ TO data item ] [
      BY data item ] [ INDEX variable ] [
      executable stmt ... ]
      [ WHILE condition [ executable stmt ... ] ]
      ENDDO

```

20 All data items and variables specified to execute a DO loop must be integer numbers. The defaults for the DO statement are: FROM = 1, TO = n, BY = 1, where n is an integer.

Conditions occur within the IF, CASEOF, and DO statements. A simple condition is:

```

expression logical operator expression
or      expression INSET Set

```

where logical operator is one of the following:

```

30      =          <=
      <>          >
      <          >=

```

A condition is built from simple conditions or other conditions:

```

35      simple condition
      or ( condition )
      or NOT condition
      or condition AND condition
      or condition OR condition
      The INSET statement is used to determine

```

-35-

whether a given variable or constant appears as a value within a SET. For example suppose X is a data item whose data type is compatible to the data type of the set:

5 X INSET SET NAME

is a condition which evaluates to either TRUE or FALSE depending on whether or not at least one of the values, in SET NAME matches the value of X.

10 A programmer uses the Rules Language statements to write out the logic of the program. For further information on the Rules Language, reference should be made to Appendices A-F attached hereto.

3. User Interface Windows

15 The user interface for any application is constructed using the CASE facilities Window Painter 8, FIG. 2. As described above, the Window Painter Module 16 is the program designed to help build user interface in a given operating system. In the example of the PC
20 environment above the rules painter would interface with the Microsoft Window Operating System. The Window Painter Module 16 would then feature the mouse-based use interface.

25 The Window Painter Module 16 is also used to create screen mappings of data designated in the Field 102 and View Entitles 100 (see FIG. 5). The Window Painter Module 16, FIG. 2, creates a file. That file contains data to create a screen mapping of the Window View. After creating and saving the panel, the CASE
30 facility of the invention also generates other files, such as for example a file that contains the code for the "painted" panel. Another file could contain the code for the menu structures used by a panel.

-36-

G. Code Generation

Once an application has been modeled, and the Rules, Components and Windows have been created, the CASE facility produces source code.

5 Preparation is the code generation phase.

When an application is prepared, the View 100 and Set 116 Entities, FIG. 5, are used to create copybooks. A copybook is a file which is copied into the source code of a program module. The CASE facility creates a
10 copybook for each data structure and includes a copy of that file in every executable Rule, Window or Component related to a particular View or Set.

Preparation also generates environment-specific source code for each Rule Language module. The
15 Rule Entity 94, FIG. 5, contains an attribute specifying the environment destination of each Rule Module. The Code Generator, 24, FIG. 2, takes high-level Rules Language statements and translates them into source code in a language supported by the hardware environment of
20 the Rule's destination. For example, if a Rule was to be executed on a Personal Computer that supported only the C language, the Code generator would translate the Rule Language Statements in C.

If the Rule Language statements specified in a
25 module can be successfully generated into source code, the code is stored in files in the Repository 4.

However, if there are logical errors in the Rule
Language statements, no code is saved and error messages detailing the unsuccessful result are saved in a Failed
30 Results File.

Once preparation has been performed, detailed reports can be made for each Entity in the entity-relationship model. Those reports are a program's technical documentation.

-37-

The reader is referred again to FIG. 2A which depicts an example of an implementation of the Code Generation Module 24, FIG. 2 in the three-tiered environments of mainframe 1, minicomputer 2, and PC 3.

5 In that example the source code for each of the Rules Language Modules, need not be generated in the hardware environment that will be each rule's destination; however, the step of compilation of that source code would occur in the destination environment of the

10 corresponding rule. In the PC environment 3, a Prepare Facility 59, FIG. 2A could prepare, for example C language source code, for each PC executable rule using the Rules Language Code, and all view fields and component windows associated with that rule module's

15 rule entity. The Prepare Facility 59 could also generate for example COBOL source code for rules designated to execute on the mainframe. In the mainframe environment 1, a mainframe front-end module 57 provides the ability to translate the Rules Language

20 Module statements to the source code language statements supported by the hardware environment of that rule's destination. The System 88 HPS front-end module 56 performs the same task in a minicomputer environment.

To perform code generation in the example, the

25 code generation module in each environment uses the rules language modules plus information needed from the entity relationship model. To transport the Rules Language Modules and generate source code modules to the different environments, the user can, using the database

30 administration module (DBA) 44, upload the module files to the Central Repository 4, and then, using separate modules 34, 46 to download the information to specific environments for compiling.

In the case of a rule designed to execute on

-38-

the mainframe environment 2 from the PC environment 3, a main view module 36 provides access to the mainframe environment 1, via the communications manager module 8, creating an emulation of a non-intelligent terminal in the PC environment.

From the PC environment 3, access to the minicomputer environment 2 in this example is provided by a Talk View Module 46, which communicates with the System 88 Front-end Module 56 using a PC connect module 50, and file translation module 48. At this stage of development, each generated module's source code statements can be examined with a Rule View Debugger associated with each respective environment. The Rule View Debugger is an aspect of the Testing Facility 10 of the present invention and is described more fully below. Successfully tested modules can be stored in the Central Repository 4.

I. Assembly

When all the source code modules of an application have been distributed to each environment, they must be assembled into a running computer program. The steps are to: 1) Compile the source code into object code; 2) Enable communication routines to allow interaction of program modules across environments; and 3) Bind or link the compiled code.

Assembly must be executed in each environment where program modules were distributed. For example, on a system incorporating PC processing, the PC will have a PC-based System Assembler 28, FIG. 2, that will enable a programmer to compile and bind the PC-based rules.

In the same example, separate assembly would have to be completed on a mini computer. For example on a IBM S/88 or Stratus mini computer, a command such as

-39-

"Build the Rule Router Application" would take the Frontier Rules and link them to every other Rule modules related to it in the mini-environment. The function also binds the code into an executable module.

- 5 Mainframe program modules are separately assembled. For example, in the exemplary IBM environment supported by the CICS operating system and DB2 relational database, a mainframe-based System Assembler 28, FIG. 2, provides assembly functions for
- 10 the Rule and Component source code. For a Rule, an assembler would:
- ~ verify that the Rule is a valid mainframe rule;
 - 15 ~ verify that the Rule has completed the code generation process;
 - ~ prompt the user to indicate if the Rule is to be setup as a Frontier Rule;
 - 20 ~ read the Bind file to load the on-line Views File and the on-line Relationship File for runtime PCI and Conversion use;
 - 25 ~ load the source code to the online Source file for rule View use;
 - ~ automatically perform the DB2 Bind based upon the relationships defined to the Repository for Frontier Rules that use Components which issue DB2 calls;
 - 30 ~ display a list of other Rules and Components affected by the setup of a particular Rule;
 - 35 ~ assign a unique Identifier and a unique DB2 Plan name for Frontier Rules that use Components which issue DB2 calls;
 - 40 ~ Rebind a DB2 Plan for a Frontier Rule when one of the Rule's related Components, that issue DB2 calls, has been modified;
 - 45 ~ remove the rule from the mainframe environment when the Rule is no longer needed;

-40-

A Component assembler in a mainframe environment would perform these functions:

- 5 ~ provide the facilities for a programmer to compile his Component in a language supported by the system;
- 10 ~ allow the programmer to view the results of his compile (compile listing and link edit map);
- 15 ~ display a list of other Rules and Components affected by the setup of a particular Component;
- ~ allow the user to edit the source code of a mainframe Component;

J. Execution

Upon successful assembly of the application modules in each hardware environment the program can now be executed and tested. The CASE facility provides the ability to execute and test the application from any environment in the hardware architecture. For example, using the hardware architecture comprised of an IBM Mainframe, an IBM Stratus mini computer and PC workstation, as in FIG. 1, a programmer seated at a PC would have the option to:

- 30 ~ Execute the PC based portion of an application; no links will be created with modules executing in Stratus or Mainframe other environment;
- 35 ~ Test the modules and modular executing in one other environment. In this case links are created with utilities other environments. This permits communication between environments;
- 40 ~ Execute or test the entire program. In this case links are created with each of the required execution environments.

-41-

K. Revising an Application

The process of changing the elements in an application using the Entity/Relationship Modeling system and the Rules Language is a straightforward process. The database is entered and the program element is changed. However, small changes in this system can have large consequences. In general any entity type that Owns, Uses, or Includes Entities that have been changed will have to be reprepared or "Modified". The CASE facility accomplishes the modification through the Software Distribution System 32, FIG. 2. (See the aforementioned International Application incorporated by reference, supra.)

15 L. Testing Facility

Finally, the CASE system provides debugging and testing facility that enables users to evaluate the performance of a application.

Generally the applications are first inputted on the computer at code level. Traditional code debugging tools are designed for testing at the code level only. Instead, the CASE facility of the present invention provides a Rule View Module for high-level debugging which is part of the Testing Facility 10 depicted in FIG. 1. A version of Rule View would exist in each of the operating environments in a hardware configuration. Applications spanning more than one environment require a separate testing for the modules in each environment.

30 Calls to the Rule View Module are automatically embedded by the CASE facility when code is generated. Rule View runs the application in a Rules-level debugger locating errors and problems that occur. It can be used interactively to step through a Rules

-42-

process and examine the contents a View Copybook at any point. Rule View gives programmers the ability to:

1. Initiate the execution of a Rule;
2. Break the interface between a Rule and a
5 Second Rule or Component;
3. Step into the logic of a Rule;
4. Step over Rule or Component Module;
5. Step back to the beginning of a Module;
6. Examine and modify any Field within a
10 View owned by the active Rule;
7. Review the active Rule's source code;
8. Save any View data for future reference
and re-use;
9. Print the Rule source code and View data.

15 Rule View will assemble at each breakpoint, a list of Views that can be examined and edited. The number of Views displayed depends on where and how Rule View is used. In most cases, the Views available for display will contain the input, output of the Rule being
20 executed. In more advanced situations -- where asynchronous Rules are being debugged or multiple applications are being tested -- many Views may be available for examination. The contents of a View can be printed or saved in a file.

25 The Rule View editor permits the programmer to edit any data by using a Field Editor. With the Field Editor, a programmer can change any data by typing over the old information. The editor also permits the user to restore the contents of any field to its original
30 value.

To review a Rule's source code, a Text Editor provides an option where the line of source code currently being executed is highlighted at all times. If multiple modules are running under Rule View, the
35 source that is displayed will be the source code that

-43-

Owens the data in the current View.

M. Software Distribution for General Use

The previous discussion of the CASE tool facility of the present invention was limited to the modeling and development of a single copy of an application program, distributed, as for example, across the three hardware environments of a PC, a minicomputer and a mainframe. A single copy of the modules comprising the program was distributed to the appropriate hardware environments, and the application was debugged and its performance was rated. However, once the application is tested and ready for general use many copies of the modules comprising the program need to be distributed and maintained. The Software Distribution System Module 32 of the present invention will distribute multiple copies of the modules of an application throughout a parallel processing hardware environment, so that many users can use the same program. For more information on the Software Distribution System see the aforementioned International Application.

N. Non-Intelligent Terminal Converse (NITC)

In addition to the hardware configuration mentioned above, the CASE facility of the present invention can be also implemented using a hardware configuration in which the ultimate user of the application program does not have access to a processing terminal, such as a PC. Instead, the end-user would have access only to a non-intelligent terminal. An example would be an IBM brand 3270 terminal connected to a mainframe computer such as the IBM 3090, specified as an exemplary embodiment above. However, other

-44-

appropriate terminals can be used when implementing the invention on other mainframe hardware environments. A non-intelligent terminal is used sometimes, because it is less expensive for a massive user group to access a single mainframe processor, rather than giving each user his or her own personal processor. In these instances, however, it might be that the programming specialists would still develop applications with a PC, using the same Development Workbench Modules 34 as described in FIG. 2A. Those development tools permit the programmer, for example, to build user interface files (i.e. window files) that are used in conjunction with the CASE facility converse function described above. Given that the end-user in the case above has no PC to converse the created windows, the graphic interface specified in the windows files would be useless.

However, the present invention provides a feature to create a graphical interface using a non-intelligent terminal converse (NITC) Module 140, FIG. 2A. The non-intelligent terminal converse module makes it possible to build applications that display graphic user interface to an end-user using, for instance, a non-intelligent device connected to a mainframe. The non-intelligent terminal converse module can also convert existing PC workstation software modules so that they can execute on a mainframe environment. By providing this function in the present invention, a mainframe executing application can be designed, constructed, debugged and prototyped completely within a PC environment, using the Development Workbench Modules 34, without accessing the mainframe until public distribution.

In the entity relationship model, the window entity is still specified for a window file that will be

-45-

used for graphic user interface on the mainframe.

In the PC environment the Development Workbench Modules 34 enable a user to paint the window objects (called panels) with which the end user of the application interacts. The Window Painter Module is
5 discussed above.

The panel can be stored in the Central Repository 4 as a file associated with the Window entity. However, these panels must be designed using
10 screen objects that can be supported on the non-intelligent terminal converse terminal device.

A window to execute on both the PC and non-intelligent terminal can be specified for example by selecting an option in the Window Painter Module 16.
15 The choice of the option converts the current Window into a format that mimics the non-intelligent terminal converse screen. For example, Fields could be translated to row/column coordinates, Pushbuttons could be moved to the bottom of the screen and require
20 keyboard equivalents. Selecting this choice also sets the Window Painter Module in a non-intelligent terminal panel building mode that prevents the placement of new objects that cannot be supported by a non-intelligent terminal converse terminal device.

25 The major difference between a PC Workstation and a non-intelligent terminal device is that the non-intelligent terminal converse device manipulates text characters, while PC Workstation manipulates graphic pixels which gives the PC Workstation a greater degree
30 of control. To account for this difference, the Window Painter Module could, for example, convert all text objects to a standard font.

The Windows created in the mainframe environment support many of the features that would be

-46-

present in PC environment Windows, including e.g. colors, editable and text Fields (i.e., for display only), Field display attributes (e.g., size or type of Field, range, etc.), Pushbuttons and other function key
5 equivalents.

Following the construction of the Application on the PC Workstation, the objects that make up the application must be uploaded to the Mainframe to be prepared for execution.

10 The run-time portion of the Non-Intelligent Terminal Converse module manages all the converse functions and provides the facilities for a developer to dynamically modify the converse functions.

 The primary function of the converse modules
15 is to manage all the non-intelligent terminal screen input and output for a Rule conversing Window. In addition, the non-intelligent terminal converse module communicates the end-user's actions to the Rule by populating a selected field with the selected text
20 string. Prior to passing the end-user's input to the invoking Rule, the converse module performs the necessary edit and validation of every input Field as specified by the attributes defined in the Repository and in the Window definition.

25 The non-intelligent terminal converse module is invoked every time a Rule converses a Window. When a Rule converses a Window, the non-intelligent terminal converse module reads a pre-determined file to obtain the physical attributes of the Window. These physical
30 attributes are used to construct the data stream commands used to display the Window on a non-intelligent terminal device.

 The non-intelligent terminal converse module performs edit and validation of all input Fields prior

-47-

to returning control to the invoking Rule. For all numerical Fields, the non-intelligent terminal converse module ensures that the data entered is numeric and adjusts the significance based upon an entered decimal point or the implied decimal point specified in a numeric picture string.

For example, if a decimal Field is defined as five digits in length with two of the digits to the right of the decimal point and edited using a numeric picture string of "9999.9", a string of 123 entered into the Field will have the following results. The decimal Field will contain the value 12.30 and the screen field will display 0012.3. In addition to using numeric picture strings to edit and validate numeric input, a range limit can be set using the Window Painter Module to ensure that the numeric value entered is not beyond the minimum and maximum values of the specified range.

Character Fields are only validated if they are the input value matches by a Set of permissible Values. This Set and its Values are defined as entities to the Central Repository. To set a Field to require a set of permissible values, the name of the Set entity must be specified in the Field's Reference File Attribute when setting the Field's Object Edit Specifications using the Window Painter Module. The non-intelligent terminal converse module will verify that the entered character string is contained in the set of permissible values.

For any fields in error, the non-intelligent terminal converse module will flag the fields in error and will display an error message. For non-intelligent devices that support color, the fields in error could be highlighted.

As stated earlier, the pushbutton objects on

-46-

present in PC environment Windows, including e.g. colors, editable and text Fields (i.e., for display only), Field display attributes (e.g., size or type of Field, range, etc.), Pushbuttons and other function key
5 equivalents.

Following the construction of the Application on the PC Workstation, the objects that make up the application must be uploaded to the Mainframe to be prepared for execution.

10 The run-time portion of the Non-Intelligent Terminal Converse module manages all the converse functions and provides the facilities for a developer to dynamically modify the converse functions.

The primary function of the converse modules
15 is to manage all the non-intelligent terminal screen input and output for a Rule conversing Window. In addition, the non-intelligent terminal converse module communicates the end-user's actions to the Rule by populating a selected field with the selected text
20 string. Prior to passing the end-user's input to the invoking Rule, the converse module performs the necessary edit and validation of every input Field as specified by the attributes defined in the Repository and in the Window definition.

25 The non-intelligent terminal converse module is invoked every time a Rule converses a Window. When a Rule converses a Window, the non-intelligent terminal converse module reads a pre-determined file to obtain the physical attributes of the Window. These physical
30 attributes are used to construct the data stream commands used to display the Window on a non-intelligent terminal device.

The non-intelligent terminal converse module performs edit and validation of all input Fields prior

APPENDIX A: RULES LANGUAGE SYNTAX**Tokens of the Rules Language**

Tokens are the atoms from which a programming language is built. All reserved words such as IF, AND, FROM, ... represent tokens of the Rules Language. Other examples are special symbols such as ")" (right parenthesis), "<=" (less or equal relational operator), and the like. Finally you will also encounter tokens such as DICT view (View Name) which denote references stemming from the Repository.

Reserved Words

| | | |
|------------|-------------|-----------|
| ABS | EVERY | OVERLAY |
| ALL | EXISTS | PC |
| AND | EXP, EXP 10 | PREV |
| ASCENDING | EXTERN | PUT |
| ASYNCH | EXTRACT | QUEUE |
| ATTACH | FALSE | REFRESH |
| AVG | FIELD | RETRIEVE |
| BEEP | FLASH | RETURN |
| BY | FORALL | RIGHTJ |
| CASE | FROM | ROUND |
| CHAR | IF | ROUTE |
| CICS | IN | RULE |
| CLEAR | INDEX | SET |
| COMPONENT | INSET | SETERROR |
| CONVERSE | INTEGER | SMALLINT |
| CURRENT | LEFTJ | SPACES |
| DCL | LENGTH | SQL |
| DELETE | LIKE | SQRT |
| DEPENDING | LOG, LOG10 | STRATUS |
| DESCENDING | MAP | STRING |
| DETACH | MAX | SUBSTRING |
| DIV | MIN | SUM |
| DO | MOD | TO |
| DOMAIN | MODULE | TRIM |
| ELSE | MOVE | TRUE |
| EMPTY | NEST | TYPE |
| ENDCASE | NEXT | USE |
| ENDDCL | NOT | VIA |
| ENDDO | NUMERIC | VIEW |
| ENDEXTERN | OCCUR | WHILE |
| ENDFORALL | OF | WINDOW |
| ENDIF | ON | ZERO |
| ENDSET | OR | ZEROES |

Reserved Symbols

-50-

| <u>Characters</u> | <u>Description</u> | <u>Symbol</u> |
|-------------------|--------------------|---------------|
| *> | Left comment | - |
| <* | Right comment | - |
| (| Left parenthesis | LP |
|) | Right parenthesis | RP |
| = | Equal | EQ |
| >= | Greater or equal | GE |
| > | Greater than | GT |
| <= | Less than or equal | LE |
| < | Less than | LT |
| <> | Not equal | NE |
| , | Comma | COMMA |
| ; | Semicolon | SEMI |

Repository Entity Identifiers

MODULE_NAME
 RULE_NAME
 WINDOW_NAME
 VIEW_NAME
 SYMBOL_NAME
 SET_NAME

PRECEDENCE TABLE

The following precedence table lists the precedence or binding power of the Rules Language operators in increasing order. Operators which occur on the same line have amongst themselves the same binding power.

| | |
|-------------------|-----------------|
| OR | left to right |
| AND | left to right |
| NOT | right to left |
| EQ NE LE LT GE GT | non-associative |
| INSET | non-associative |
| MINUS | left to right |
| IN | non-associative |
| OF | right to left |

The right column describes the associativity of the operations. We can see from the "AND line" of the precedence table that it is perfectly legal to write for conditions cond1, cond2, cond3

cond1 AND cond2 OR cond3

and that implied order or parsing is first

cond1 AND cond2

and afterwards

cond2 AND cond3

Another example: Given a partial qualification

V1 OF V2 OF V3

involving three Views V1, V2, V3 then the parser first recognizes

V2 OF V3

and afterwards

V1 OF V2.

Reserved Symbols

-50-

| <u>Characters</u> | <u>Description</u> | <u>Symbol</u> |
|-------------------|--------------------|---------------|
| *> | Left comment | - |
| <* | Right comment | - |
| (| Left parenthesis | LP |
|) | Right parenthesis | RP |
| = | Equal | EQ |
| >= | Greater or equal | GE |
| > | Greater than | GT |
| <= | Less than or equal | LE |
| < | Less than | LT |
| <> | Not equal | NE |
| , | Comma | COMMA |
| ; | Semicolon | SEMI |

Repository Entity Identifiers

MODULE_NAME
 RULE_NAME
 WINDOW_NAME
 VIEW_NAME
 SYMBOL_NAME
 SET_NAME

PRECEDENCE TABLE

The following precedence table lists the precedence or binding power of the Rules Language operators in increasing order. Operators which occur on the same line have amongst themselves the same binding power.

| | |
|-------------------|-----------------|
| OR | left to right |
| AND | left to right |
| NOT | right to left |
| EQ NE LE LT GE GT | non-associative |
| INSET | non-associative |
| MINUS | left to right |
| IN | non-associative |
| OF | right to left |

The right column describes the associativity of the operations. We can see from the "AND line" of the precedence table that it is perfectly legal to write for conditions cond1, cond2, cond3

cond1 AND cond2 OR cond3

and that implied order or parsing is first

cond1 AND cond2

and afterwards

| | | |
|----------------|--|--|
| | | -53- |
| | | do_idx_stmt |
| assign_stmt: | | MAP expr TO variable |
| overlay_stmt: | | OVERLAY dat_item TO variable |
| clear_stmt: | | CLEAR variable |
| use_stmt: | | USE MODULE MODULE_NAME |
| | | USE RULE RULE_NAME nest clause |
| nest_clause: | | (empty) |
| | | NEST |
| return_stmt: | | RETURN |
| converse_stmt: | | CONVERSE window_clause |
| window_clause: | | WINDOW WINDOW_NAME |
| async_stmt: | | ASYNc attach_detach RULE_NAME VIA RULE |
| | | RULE_NAME |
| | | ASYNc ROUTE RULE RULE_NAME TO RULE |
| | | RULE_NAME |
| | | ASYNc refresh_stmt |
| attach_detach: | | ATTACH |
| | | DETACH |
| refresh_stmt | | REFRESH window_clause |
| | | field_clause |
| | | view_clause |
| | | occur_clause |
| | | beep_clause |
| | | flash_clause |
| view_clause: | | (empty) |
| | | VIEW data_item |
| field_clause: | | (empty) clause |
| | | FIELD data_item |
| occur_clause: | | (empty) |
| | | OCCUR data_item |
| beep_clause: | | (empty) |
| | | BEEP |
| flash_clause: | | (empty) |
| | | FLASH |
| cond_stmt: | | if_stmt |
| | | case_of-stmt |
| | | |

| | |
|----------------|--|
| if-stmt: | IF cond stmt-list ENDIF |
| | IF cond stmt_list ELSE stmt_list ENDIF |
| case_of_stmt: | CASE_OF caseof_var case_list ENDCASE |
| | CASE_OF caseof_var case_list CASE OTHER |
| | stmt_list ENDCASE |
| caseof_var: | variable |
| case_list: | single_case |
| | case_list single_case |
| single_case: | CASE case_lit_list stmt_list |
| case_lit_list: | lit |
| | set_const |
| | MINUS int_lit |
| | MINUS dec_lit |
| | case_lit_list lit |
| | case_lit_list set_const |
| do_stmt: | DO stmt_list WHILE cond stmt_list ENDDO |
| | DO stmt_list ENDDO |
| do_idx_stmt: | DO do_clauses stmt_list while_clause ENDDO |
| do_clauses: | from_clause to_clause by_clause |
| | index_clause |
| from_clause: | (empty) |
| | FROM expr |
| to_clause: | |
| | TO expr |
| by_clause: | |
| | BY expr |
| index_clause: | |
| | INDEX variable |
| while_clause: | |
| | WHILE cond stmt_list |
| cond: | simple_cond |
| | LP cond RP |
| | not cond %prec NOT |
| | cond AND cond %prec AND |
| | cond OR cond %prec OR |
| simple_cond: | expr rel_op expr |
| | expr INSET SET_NAME |
| rel_op: | EQ |
| | NE |
| | num_rel_op |

num_rel_op: LT -55-
 |
 LE
 GT
 GE

variable: simple-var
 |
 qual-id qual_var_list
 simple_var subscr_unit
 qual_id qual_var_list subscr_unit

subscr_unit: LP item_list RP

item_list: expr
 |
 expr COMMA expr
 expr COMMA expr COMMA expr

qual_var_list: OF VIEW_NAME
 |
 qual_var_list OF VIEW_NAME

dat_item: variable
 |
 lit
 set_const

set_const: SYMBOL_NAME
 |
 SYMBOL_NAME IN SET_NAME

lit: char_lit
 |
 int_lit
 dec_lit

| | |
|----------------|--|
| if-stmt: | IF cond stmt-list ENDIF |
| | IF cond stmt_list ELSE stmt_list ENDIF |
| case_of_stmt: | CASE_OF caseof_var case_list ENDCASE |
| | CASE_OF caseof_var case_list CASE OTHER |
| | stmt_list ENDCASE |
| caseof_var: | variable |
| case_list: | single_case |
| | case_list single_case |
| single_case: | CASE case_lit_list stmt_list |
| case_lit_list: | lit |
| | set_const |
| | MINUS int_lit |
| | MINUS dec_lit |
| | case_lit_list lit |
| | case_lit_list set_const |
| do_stmt: | DO stmt_list WHILE cond stmt_list ENDDO |
| | DO stmt_list ENDDO |
| do_idx_stmt: | DO do_clauses stmt_list while_clause ENDDO |
| do_clauses: | from_clause to_clause by_clause |
| | index_clause |
| from_clause: | (empty) |
| | FROM expr |
| to_clause: | |
| | TO expr |
| by_clause: | |
| | BY expr |
| index_clause: | |
| | INDEX variable |
| while_clause: | |
| | WHILE cond stmt_list |
| cond: | simple_cond |
| | LP cond RP |
| | not cond %prec NOT |
| | cond AND cond %prec AND |
| | cond OR cond %prec OR |
| simple_cond: | expr rel_op expr |
| | expr INSET SET_NAME |
| rel_op: | EQ |
| | NE |
| | num_rel_op |

-57-

| Symbol | Name | Function |
|--------|---------------------------------|--|
| (| Left parenthesis | Grouping; subscripts |
|) | Right parenthesis | |
| = | Equal symbol | To form symbols for relational operators |
| < | Less than symbol | |
| > | Greater than symbol | |
| <> | Not equal symbol | Test for inequality |
| <= | Less than or equal to symbol | Test for less than or equal to |
| >= | Greater than or equal to symbol | |
| - | Minus symbol | Negation |
| , | Comma | Separation of list element and subscripts |
| ; | Semicolon | |
| | | Termination of declaration of local fields |
| + | Addition | For future use |
| * | Multiplication | " " " |
| / | Division | " " " |
| ++ | Union | " " " |
| ** | Intersection | " " " |
| -- | Set difference | " " " |
| << | Sub-Set | " " " |
| >> | SuperSet | " " " |

Figure 2 Binay and Unary Rules Language Operators

Comments

-58-

Comments are enclosed between "***<**" and "***>**" as delimiters. Comments cannot be nested. There is no limit to the length of a comment.

```

*> This is an example of a comment <*>
*> This is <*>another example <***> of a comment <*>

*>

* It is
* prefecation OK to
* spread a comment
* over more than
* one line
* and use * or < on their own
<*>

```

Input past column 72 is treated as a comment (i.e., ignored).

Data Items

Data items are the variable and constants of the Rules Language. Fugure 6-5 shows how these items are related to one another.

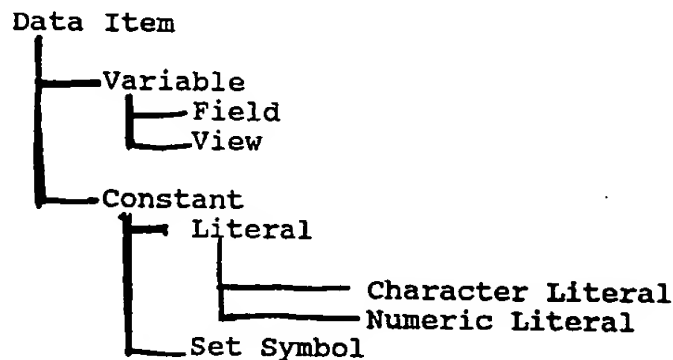


Figure 3 Data Item Hierarchy Table

Variables

Variables are defined using the View and Field HPS entities. Fields and Views are defined and described outside of the Language itself using the facilities of the Repository. Note that, with certain restrictions, Fields and Views can be declared local to Rule Block by means of the DCL construct!

Fields

-59-

A Field is a variable which behaves like an atom; that is, it cannot be divided into smaller units. With the exception of a limited facility for declaring fields local to a Rule program, Fields are defined using the facilities of the HPS Repository. They are not defined within a Rule. Some examples of Fields and their types follow.

FLD_1 SMALLINT

WORD_COUNT INTEGER

In the example given above, FLD_1 is a two byte (signed) integer variable and WORD_COUNT is a four byte (signed) integer variable.

FLD_2 CHAR

CUSIP_DESCR CHAR (20)

FLD_2 and CUSIP_DESCR are character Fields of lengths 1 and 20, respectively.

CUST_BAL_AMT DECIML (15, 2)

CUST_BAL_AMT is a "Dollars and Cents" variable with up to 15 digits precision, two of which are to the right side of an implied decimal point. DECIMAL (p,q) Fields are signed quantities.

CASH_TXN_BAL_AMT PIC S9999V99

CASH_TXN_BAL_AMT and CASH_TXN_CR_AMT are both numeric variables with up to $4+2 = 6$ digits precision, two of which are to the right side of an implied (V="virtual") decimal point. The S in the first example denotes an optional sign whereas in the declaration:

CASH-TXN_CR_AMT PIC 9999V99

CASH_TXN_CR_AMT cannot be negative.

TEMP_BUFFER VARCHAR (50)

The constructs of the Rules Language do not distinguish TEMP_BUFFER from a CHAR (50) variable. Both CHAR (50) and VARCHAR (50) allocate 50 bytes of storage and in both cases anything that is moved into either of them will be left justified and padded to the right with spaces. A VARCHAR (n) variable keeps track of its "actual" length through the use of a separate length Field which is completely transparent to the Rules Language programmer.

The following table summaries the Field types which are supported within the HPS Rules Language and some of their properties:

Comments

-58-

Comments are enclosed between "***<**" and "***>**" as delimiters. Comments cannot be nested. There is no limit to the length of a comment.

```

*> This is an example of a comment <*
*> This is <* *>another example <*> of a comment <*

*>

* It is
* prefecction OK to
* spread a comment
* over more than
* one line
* and use * or < on their own
<*
```

Input past column 72 is treated as a comment (i.e., ignored).

Data Items

Data items are the variable and constants of the Rules Language. Figure 6-5 shows how these items are related to one another.

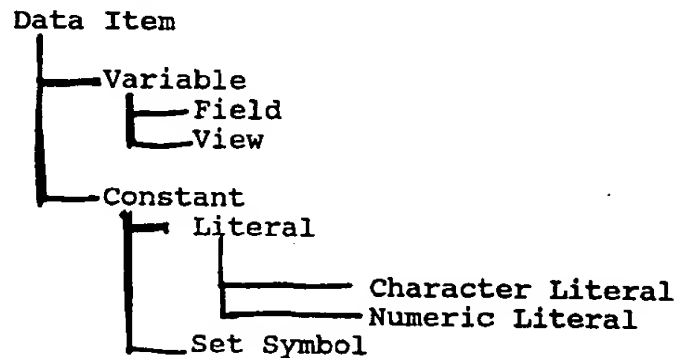


Figure 3 Data Item Hierarchy Table

Variables

Variables are defined using the View and Field HPS entities. Fields and Views are defined and described outside of the Language itself using the facilities of the Repository. Note that, with certain restrictions, Fields and Views can be declared local to Rule Block by means of the DCL construct!

-61-

SUB_VIEW, CG_BIT. The child SUB_VIEW of CG_VW_13 has the Fields CG_VARCHAR and CG_CHAR, but no subordinate Views.

```

VW_1,
    CG_VW_11
        CG_DATE      CHAR      (08),
        CG_VARCHAR   VARCHAR   (10),
        CG_CHAR      CHAR      (10),
        CG_BIT       CHAR      (01),

CG_VW-13,
    CG_DATE      CHAR      (08),
    SUB_VIEW     (05),      *>occurs 5 times<
    CG_VARCHAR   VARCHAR   (10),
    CG_CHAR      CHAR      (10),
    CG_BIT       CHAR      (01),

    CUSIP_DESCR  CHAR (20),
    CUST_BAL_AMT DECIMAL (15,2),
    CASH_TXN_BAL_AMT PIC S9999V99,

CG-VW-10
    SUB_VIEW     (05),      *>occurs 5 times<
    CG_VARCHAR   VARCHAR   (10),
    CG_CHAR      CHAR (10),
    INT_RATE     DECIMAL   (7,7),
    CG_BIT       CHAR      (01),

```

Figure 5 Data Structure From View

The above View, VW_1, illustrates many of the concepts underlying HPS data structures.

Each of the sub-Views belonging to the above tree structure is itself a tree structure of which it becomes the 01-level View. For example, CG_VW_10 defines a tree with CG_VW_10 as the top node, SUB_VIEW, INT_RATE and CG_BIT as its 2nd level leaves and nodes, and CG_VARCHAR AND CG_CHAR as its 3rd level leaves.

A view is uniquely determined within the Repository through the name of its 01-level node. In other words, VW_1 stands for the whole collection of all the Views and Fields in the figure.

Views and Fields can occur more than once within a tree. But they are not allowed to form recursive constructs by referring to themselves, either directly or through a chain of intermediate child or parent Views.

For example, CG_CHAR is a Field and SUB_VIEW is a View each appearing more than once within VW_1.

-62-

Note that in accordance with Paragraph 1, SUB_VIEW defines the same tree structure regardless of whether it appears as a child of CG_VW 10 or of CG_VW_13.

```

SUB_VIEW      (05),
  CG_VARCHAR  VARCHAR  (10),
  CG-CHAR     CHAR     (10),

```

Figure 6 Sub-View Data Structure

The Field CG_CHAR that belongs to the View SUB_VIEW that belongs to the View CG_VW_10 can contain different data from the Field CG_CHAR that belong to SUB_VIEW that belongs to CG_VW-13. In fact, they occupy different storage locations. furthermore, CG_CHAR appears elsewhere--it is also a Component of CG_VW_11. How do you tell these instances apart within a Rules program?

First, write the fully qualified name of the ambiguously determined items (which may be either Fields or Views). The fully qualified name begins with the name of the item at the lowest level (which may be either a Field or a View) and works up to the top of the tree naming each of the Views (not Fields), separating each of the names with the reserved word OF. (This naming convention is borrowed from COBOL and is different from the one used in C and PL/I. For example, in the case of CG_CHAR we have

```

CG_CHR OR SUB_VIEW OF CG_VW_11 OF VW_1
CG_CHAR OF SUB_VIEW OF CG_VW-13 of VW_1, and
CG_CHAR OF CG_VW_10 OF VW_1

```

Figure 7 Fully Qualified Names

The general notation is as follows:

Let X be a Field or View which has a parent V1 which has a parent V2 ... and so on.

Let Vn denote the top node ...

Then, the fully qualified name of X is:

X of V1 OF V2 ... OF Vn.

These fully qualified names are sufficient to discriminate between the instances of CG_CHAR. In fact, they contain redundant information for the Rules code generator to discriminate between instances. To arrive at the necessary and sufficient information, simply delete the names of the Views which are common to the fully qualified names. Of course, do not delete the name at the base of the upward path to the top level View. Applying this to the above example results in:

-63-

| | |
|---------|------------------|
| CG_CHAR | OF CG_VW_11 |
| CG_CHAR | OF CG_VW-13, and |
| CG_CHAR | OF CG_VW_10 |

Figure 8 Edited Qualified Names

This is the minimal information which the Rule code generator requires to discriminate between the instances of CG_CHAR. Furthermore, this is also the minimal information your eye would need to tell the instances apart with a glance at an outline of VW_1.

There is, however, one type of ambiguity which cannot be resolved with fully qualified names if partially qualified names are permitted. Consider the following View structure:

```

VIEW10
├── FIELD 1,      *>first occurrence of FIELD1 < *
├── VIEW1,
│   ├── FIELD1    *>second occurrence of FIELD1 < *
│   ├── FIELD2,  *>first occurrence of FIELD2 < *
│   └── VIEW2,
│       └── FIELD2,  *>first occurrence of FIELD2 < *

```

FIGURE 9 AMBIGUOUS FIELD STRUCTURE

VIEW0 contains FIELD1 as an ambiguous reference because the only sensible choices for (partially qualified) names for the first occurrence are either:

FIELD1

or

FIELD1 OF VIEW 10,

but both of them also make reference to the second occurrence of FIELD1. Note in contrast that one can clearly make a distinction between the contents of:

FIELD2 OF VIEW1,

and

FIELD2 OF VIEW2.

Subscripts

The Rules Language supports views with subscripts (OCCURS Attribute of the View-View/HPS relationship). They are the counterpart of tables in COBOL and arrays in C or PL/1.

Assume that VW-2 is a View defined in the HPS Repository as follows:

-62-

Note that in accordance with Paragraph 1, SUB_VIEW defines the same tree structure regardless of whether it appears as a child of CG_VW_10 or of CG_VW_13.

```

SUB_VIEW      (05),
  CG_VARCHAR   VARCHAR   (10),
  CG-CHAR      CHAR      (10),

```

Figure 6 Sub-View Data Structure

The Field CG_CHAR that belongs to the View SUB_VIEW that belongs to the View CG_VW_10 can contain different data from the Field CG_CHAR that belong to SUB_VIEW that belongs to CG_VW-13. In fact, they occupy different storage locations. furthermore, CG_CHAR appears elsewhere--it is also a Component of CG_VW_11. How do you tell these instances apart within a Rules program?

First, write the fully qualified name of the ambiguously determined items (which may be either Fields or Views). The fully qualified name begins with the name of the item at the lowest level (which may be either a Field or a View) and works up to the top of the tree naming each of the Views (not Fields), separating each of the names with the reserved word OF. (This naming convention is borrowed from COBOL and is different from the one used in C and PL/I. For example, in the case of CG_CHAR we have

```

CG_CHR OR SUB_VIEW OF CG_VW_11 OF VW_1
CG_CHAR OF SUB_VIEW OF CG_VW-13 of VW_1, and
CG_CHAR OF CG_VW_10 OF VW_1

```

Figure 7 Fully Qualified Names

The general notation is as follows:

Let X be a Field or View which has a parent V1 which has a parent V2 ... and so on.

Let Vn denote the top node ...

Then, the fully qualified name of X is:

X of V1 OF V2 ... OF Vn.

These fully qualified names are sufficient to discriminate between the instances of CG_CHAR. In fact, they contain redundant information for the Rules code generator to discriminate between instances. To arrive at the necessary and sufficient information, simply delete the names of the Views which are common to the fully qualified names. Of course, do not delete the name at the base of the upward path to the top level View. Applying this to the above example results in:

-65-

01 03 05 07

the highest level number supported is 39.

Constants

Constants appear in the Rules Language in two manifestations:

- . literals, and
- . Set symbols.

They can represent:

- . characters, such as "New York City," or
- . (signed) integer numbers, such as 123 or 1234567, or
- . (signed) decimal numbers such as 123.45 or 1234.50.

Numeric Literals

There are two types of numeric literals: integer numbers and decimal numbers. An integer is a sequence of one or more digits. A decimal number is a sequence of one or more digits followed by a decimal point which may be followed by zero or more digits. A negative number is expressed by preceding the digits with a minus sign.

For example, 0 and 123 are integers. 123.0, -7734.33 and 123.456 are valid decimal constants as is 0.456.

These literals have restrictions upon the number of digits they contain depending upon their type (integer or decimal). Integers consist of up to 15 digits while decimals contain up to 16 digits (15 + 1 for the decimal point). Positive numbers may not be preceded by a plus sign but negative quantities must be preceded by a minus sign.

Character Literals

A character literal is a '(single quote) followed by zero to 50 character (other than ') followed by ' a'. The following are valid character literals.

'This is a valid character constant'

'ZYZZY and PLUGH are magic words'

The last example is a null string. If it is necessary to include a single quote character in the literal itself, the usual dodge of using two consecutive single quotes may be employed. For example,

-66-

'Mike' 's computer is broken.'

is

Mike's computer is broken

Note that single quotes must be used. Double quotes are not valid.

Sets and Symbols

A set is a collection of constant value data items. All constants are of the same data type. For example, all of them will be CHAR(6) or all of them will be SMALLINT. Because of this property we can refer to the data type of a Set. The constants of a Set behave like Fields rather than views in the following sense:

- . They cannot be broken down into lower levels and
- . they cannot have an Occurs clause.
- . In addition, the whole Set does not have an Occurs clause.

Example:

| | | |
|-----|----------|-----------|
| SET | MONTHSET | SMALLINT, |
| | JAN | VALUE 1, |
| | FEB | VALUE 2, |
| | MAR | VALUE 3, |
| | APR | VALUE 4, |
| | MAY | VALUE 5, |
| | JUN | VALUE 6, |
| | JUL | VALUE 7, |
| | AUG | VALUE 8, |
| | SEP | VALUE 9, |
| | OCT | VALUE 10, |
| | NOV | VALUE 11, |
| | DEC | VALUE 12; |

The example defines a Set called MONTHSET which is composed of a collection of twelve constants which represent the twelve months of a year. Their symbols are JAN, FEB, ..., DEC and their associated values are integer literals 1, 2, ..., 12.

MONTHSET is of the type SMALLINT which means the format of the values is SMALLINT. There cannot be a VALUE 3.21 nor VALUE "NOT AN SMALLINT" nor VALUE 1234565. This last example is invalid because an SMALLINT cannot exceed the number 32767).

This example will be used for further illustration of the Set concepts.

Each of the constants of a set possesses an identifier called a SYMBOL, and a value. The SYMBOL is a vehicle for referencing

-66-

'Mike' 's computer is broken.'

is

Mike's computer is broken

Note that single quotes must be used. Double quotes are not valid.

Sets and Symbols

A set is a collection of constant value data items. All constants are of the same data type. For example, all of them will be CHAR(6) or all of them will be SMALLINT. Because of this property we can refer to the data type of a Set. The constants of a Set behave like Fields rather than views in the following sense:

- . They cannot be broken down into lower levels and
- . they cannot have an Occurs clause.
- . In addition, the whole Set does not have an Occurs clause.

Example:

| | | |
|-----|----------|-----------|
| SET | MONTHSET | SMALLINT, |
| | JAN | VALUE 1, |
| | FEB | VALUE 2, |
| | MAR | VALUE 3, |
| | APR | VALUE 4, |
| | MAY | VALUE 5, |
| | JUN | VALUE 6, |
| | JUL | VALUE 7, |
| | AUG | VALUE 8, |
| | SEP | VALUE 9, |
| | OCT | VALUE 10, |
| | NOV | VALUE 11, |
| | DEC | VALUE 12; |

The example defines a Set called MONTHSET which is composed of a collection of twelve constants which represent the twelve months of a year. Their symbols are JAN, FEB, ..., DEC and their associated values are integer literals 1, 2, ..., 12.

MONTHSET is of the type SMALLINT which means the format of the values is SMALLINT. There cannot be a VALUE 3.21 nor VALUE "NOT AN SMALLINT" nor VALUE 1234565. This last example is invalid because an SMALLINT cannot exceed the number 32767).

This example will be used for further illustration of the Set concepts.

Each of the constants of a set possesses an identifier called a SYMBOL, and a value. The SYMBOL is a vehicle for referencing

-67-

the underlying value. A Set symbol can be used in the same way as a Field. The principal difference is that the value of a Set symbol is constant and can never be altered through a Rules statement, whereas the value of a Field can be changed; for example, by clearing it or making it the target of a MAP or OVERLAY Statement.

Assume SYMXXX is a symbol belonging to a Set SETYYY. It can be referenced in either of the following ways:

SYMXXX

SYMXXX IN SETYYY

The IN keyword was chosen instead of the OF keyword to avoid overloading the meaning of the latter with too many different uses. SYMXX must be qualified with its Set if SYMXX is used, within the Rule, either as the symbol of another Set or as a Field name or as a View name.

Referring back to the MONTHSET example:

```
MAR
MAR IN MONTHSET
3
```

note that each have exactly the same meaning, namely the number three

```
MAP    FEB IN MONTHSET          TO XXXVAR OF YYYVIEW
```

has the effect of moving the value 2 into the Field XXXVAR which is, hopefully, properly declared somewhere within the Rule or its bind file as a numeric sub-Field of the View YYYVIEW.

Within a Set, there cannot be two or more symbols with the same symbolName. Note though that a Set can possess several symbols with one and the same value!

Other Identifiers

Besides data items, the following entities are used within a Rule:

- . Rule Name
- . Component Name
- . Window Name
- . Set Name

They will be described in detail during the discussion of the executable statements where they occur.

Figure 10 Example DCL...ENDDCL STATEMENT

Note that it is not possible to directly define a View within a DCL ... ENDDCL statement. It is possible to do so indirectly with the LIKE clause. In the above example, TEMP_VIEW is a temporary local View with nearly the same structure as RTAXCMPI. The only difference is that the 01-level name is TEMP_VIEW rather than RTAXCMPI and the TEMP_VIEW OCCURS 5 times.

APPENDIX D: Executable Statements of Rules Language**Assignment of Value**

Three types of statements alter the value of data contained in Fields and Views: MAP, CLEAR, and OVERLAY. The syntax and semantics of each will be described in turn.

MAP STATEMENT**Syntax**

The syntax of the MAP statement is:

MAP data_item TO variable,

where data_item is a constant, a Field name, or a View name and variable is either a Field name or a View name. Recall that the Field or View names must be unambiguously qualified and these names may include subscripts.

Some examples follows:

MAP 'Syntax' TO CG_CHAR OF CG_VW_11

MAP CG_CHAR OF CG_VW_11 TO CG_CHAR OF SUB_VIEW OF
CG_VW_13(3)

MAP 23400.00 TO CUST_BAL_AMT

Figure 1 Example Map Statements

Semantics

The results of mapping A to B, as an example, can vary greatly depending on the data types of A and B. One can see how those assignments work by studying the following matrix built according to the following principle:

The possible sources (A) for a MAP statement constitute the rows of the matrix and the possible destinations (B) constitute its columns. The syntactic correctness or result of the MAP operation is given at the intersection of the row and column. The numbers in parentheses refer to the numbered paragraphs immediately following the matrix. VW(5) and VW(8) are two Views--both with the multiple occurrences but one with fewer (5) and the other with more (8).

-71-

| Source (A) | (B) Destination | | | | |
|---------------|-----------------|-------|-------|----------------|----------------|
| | View | Field | Const | VW(5) | VW(8) |
| View | OK (1) | ERROR | ERROR | WARNING (3) | WARNING (3) |
| Field | ERROR | (4) | ERROR | ERROR | ERROR |
| Num lit | ERROR | (6) | ERROR | ERROR | ERROR |
| Char lit | ERROR | (5) | ERROR | ERROR | ERROR |
| Set | ERROR (2) | ERROR | ERROR | ERROR | ERROR |
| Symbol | ERROR | (6) | ERROR | ERROR | ERROR |
| VW(5) | WARNING | ERROR | ERROR | OK | WARNING |
| VW(8) | WARNING (3) | ERROR | ERROR | WARNING (3) | OK |

Figure 2 The Map Operation

- (1) Mapping a View to a View means mapping all sub-Views and/or Fields with corresponding names which are subordinate to the source View and the destination View. For example, assume we have:

| | |
|-----------------|----------------|
| VIEW_1 | VIEW_2 |
| ABC | OPQ |
| DEF occurs (50) | ABC |
| OPQ | DEF occurs (9) |
| XXX | BBB |
| BBB | ZZZ |
| XXY | |
| XYZ | |

where the specifics of the sub-Views and the Fields of VIEW1 and VIEW2 do not matter. Then, MAP VIEW_1 TO VIEW_2 will have the following effect:

| | | |
|-------------------|---------------|-------------------|
| ABC OF VIEW_1 | is moved into | ABC OF VIEW_2 |
| OPQ of VIEW_1 | is moved into | OPQ OF VIEW_2 |
| BBB OF VIEW_1 | is moved into | BBB OF VIEW_2 |
| DEF OF VIEW_1 (1) | is moved into | DEF OF VIEW_2 (1) |
| DEF OF VIEW_1 (2) | is moved into | DEF OF VIEW_2 (2) |

APPENDIX D: Executable Statements of Rules Language**Assignment of Value**

Three types of statements alter the value of data contained in Fields and Views: MAP, CLEAR, and OVERLAY. The syntax and semantics of each will be described in turn.

MAP STATEMENT**Syntax**

The syntax of the MAP statement is:

MAP data_item TO variable,

where data_item is a constant, a Field name, or a View name and variable is either a Field name or a View name. Recall that the Field or View names must be unambiguously qualified and these names may include subscripts.

Some examples follows:

MAP 'Syntax' TO CG_CHAR OF CG_VW_11

MAP CG_CHAR OF CG_VW_11 TO CG_CHAR OF SUB_VIEW OF
CG_VW_13(3)

MAP 23400.00 TO CUST_BAL_AMT

Figure 1 Example Map Statements

Semantics

The results of mapping A to B, as an example, can vary greatly depending on the data types of A and B. One can see how those assignments work by studying the following matrix built according to the following principle:

The possible sources (A) for a MAP statement constitute the rows of the matrix and the possible destinations (B) constitute its columns. The syntactic correctness or result of the MAP operation is given at the intersection of the row and column. The numbers in parentheses refer to the numbered paragraphs immediately following the matrix. VW(5) and VW(8) are two Views--both with the multiple occurrences but one with fewer (5) and the other with more (8).

MAP A(1) TO B

-73-

or

MAP A TO B(1)

- (4) See the Field-to-Field MAP in Figure 2. A source literal of type INTEGER or DECIMAL behaves the same as a variable of that type.
- (5) See the Field-to-Field MAP in Figure 2. A source literal of the form 'ABCDEFGH' behave the same as a variable of type CHAR(8).
- (6) See the Field-to-Field MAP in Figure 2. A source which is a symbol belonging to a Set of a given type behaves in the same way as a variable of the same type.

Mapping of VARCHAR Fields

The following describes how MAP statements operate with variables of type VARCHAR.

The major difference between variables of type VARCHAR (nn) and those of type CHAR (nn) is due to the fact that VARCHAR (nn) variables have an associated length field, named xxx_LEN for a VARCHAR variable xxx. xxx_LEN contains the "actual length" of xxx while nn is the "maximal length" of xxx.

Assume that B is a variable of type VARCHAR with length B_LEN and that A is a variable, a literal, a symbol of type VARCHAR, or of type "char." B_LEN is determined from the length of A in a MAP A TO B statement as follows.

If length of A ≤ maximum length of B, then:

B_LEN = length of A
 contents of B = contents of A padded with spaces to the right

If length of A > maximum length of B, then:

B_LEN = maximal length of B
 contents of B = first "max length of B" characters of A

Examples of the Use of VARCHAR Variables

Assume the following:

-74-

| | |
|------------|----------------------|
| CHAR_VAR1 | of type CHAR (10) |
| CHAR_VAR2 | of type CHAR (20) |
| VARCH_VAR1 | of type VARCHAR (15) |
| VARCH_VAR1 | of type VARCHAR (20) |

Assume also that we have a Rule which consists of the following MAP statements:

| | | |
|----|-----------------------------|----------------|
| 1. | Map 'ABC ' | TO CHAR_VAR_1 |
| 2. | MAP '** MY LENGTH IS 20 **' | TO CHAR_VAR_2 |
| 3. | MAP 'ABC ' | TO VARCH_VAR_1 |
| 4. | MAP CHAR_VAR_1 | TO VARCH_VAR_1 |
| 5. | MAP CHAR_VAR_2 | TO VARCH_VAR_1 |
| 6. | MAP CHAR_VAR_2 | TO VARCH_VAR_2 |
| 7. | MAP VARCH_VAR_2 | TO VARCH_VAR_2 |
| 8. | MAP VARCH_VAR_1 | TO VARCH_VAR_2 |

The following results will occur in the VARCHAR variables

| | |
|----------|--|
| #1 and 2 | Nothing has yet been assigned to VARCH- VAR_1_LEN and VARCH_VAR_2_LEN |
| #3 | VARCH_VAR_1_LEN = 5 = length of 'ABC' |
| #4 | VARCH_VAR_1_LEN = 10 = length of CHAR_VAR_1, |
| #5 | VARCH_VAR_1_LEN = 15 = length of CHAR_VAR_1, contents of VARCH_VAR_1 will be ' ** MY LENGTH IS' |
| #6 | VARCH_VAR_1_LEN = 20 = length of VARCH_VAR_1, contents of VARCH_VAR_1 will be ' ** MY LENGTH IS' 20' |
| #7 | VARCH_VAR_1_LEN = 15 = length of VARCH_VAR_1, contents of VARCH_VAR_1 will be ' ** MY LENGTH IS' |
| #8 | VARCH_VAR_1_LEN = 15 = length of VAR_VAR_1, contents of VARCH_VAR_1 will be ' ** MY LENGTH IS' |

CLEAR Statement

The CLEAR statement sets numeric Fields to zero and character Fields to blank.

Syntax

The CLEAR statement has the form:

CLEAR variable

where "variable" is fully or partially qualified name of either a View or a Field. If the variable being cleared has

-75-

subscripts, that is, multiple occurrences of any of its Views or sub-Views, then subscripts may also appear. For example,

CLEAR VW_1

will zero all the numeric Fields of VW_1 and blank the character Fields. Recall that numeric Fields include INTEGER (n), DECIMAL (p, q), and Fields described with PIC elements S, 9, and V. Character Fields include CHAR (n), VARCHAR (n).

Semantics

A Field of underlying type "Character" (not including PICs) is set to blanks. A Field of underlying type "Numeric" (including decimals and PICs) is set to zero.

A View V, say, is treated as follows:

(1) Assume that V owns Fields only but no View. Then each one of those Fields is Set to spaces or to zero, depending on whether its underlying type is "Character" or "Numeric." If V is a View that is subscripted (the corresponding OCCURS clauses could possibly be specified on a higher level than that of V), then clearing V means the same as clearing each v(S1) or v(S1, S2) or v(S1, S2 S3). Here S1, S2 and S3 denote the Subscripts of V and it depends on the number of OCCURS above and including the level of V, regardless of whether V is subscripted once, twice, or three times.

(2) If V has one or more views V1, V2, ..., as children, then treat V as follows: Clear the Fields which are children of V, just as was outlined above. Then check each V1, V2 to determine whether or not it also contains Fields as children and, if so, treat them according to (1) above. Otherwise examine each sub-View.

The net result of the above algorithm is as follows. Any View is ultimately partitioned into a Set of Fields. Each of those Fields is set to spaces or to zero, depending on whether its underlying type is "Character" or "Numeric."

OVERLAY Statement

This statement is used to replace ("overlay") one structure in storage with the content of another.

Syntax

The OVERLAY statement has the form

OVERLAY data item TO variable

where data item is a constant, Field, or View and variable is either the name of a View or a Field.

-74-

| | |
|------------|----------------------|
| CHAR_VAR1 | of type CHAR (10) |
| CHAR_VAR2 | of type CHAR (20) |
| VARCH_VAR1 | of type VARCHAR (15) |
| VARCH_VAR1 | of type VARCHAR (20) |

Assume also that we have a Rule which consists of the following MAP statements:

| | | |
|----|-----------------------------|----------------|
| 1. | Map 'ABC ' | TO CHAR_VAR_1 |
| 2. | MAP '** MY LENGTH IS 20 **' | TO CHAR_VAR_2 |
| 3. | MAP 'ABC ' | TO VARCH_VAR_1 |
| 4. | MAP CHAR_VAR_1 | TO VARCH_VAR_1 |
| 5. | MAP CHAR_VAR_2 | TO VARCH_VAR_1 |
| 6. | MAP CHAR_VAR_2 | TO VARCH_VAR_2 |
| 7. | MAP VARCH_VAR_2 | TO VARCH_VAR_2 |
| 8. | MAP VARCH_VAR_1 | TO VARCH_VAR_2 |

The following results will occur in the VARCHAR variables

| | |
|----------|--|
| #1 and 2 | Nothing has yet been assigned to VARCH- VAR_1_LEN and VARCH_VAR_2_LEN |
| #3 | VARCH_VAR_1_LEN = 5 = length of 'ABC' |
| #4 | VARCH_VAR_1_LEN = 10 = length of CHAR_VAR_1, |
| #5 | VARCH_VAR_1_LEN = 15 = length of CHAR_VAR_1, contents of VARCH_VAR_1 will be ' ** MY LENGTH IS' |
| #6 | VARCH_VAR_1_LEN = 20 = length of VARCH_VAR_1, contents of VARCH_VAR_1 will be ' ** MY LENGTH IS' 20' |
| #7 | VARCH_VAR_1_LEN = 15 = length of VARCH_VAR_1, contents of VARCH_VAR_1 will be ' ** MY LENGTH IS' |
| #8 | VARCH_VAR_1_LEN = 15 = length of VAR VAR_1, contents of VARCH_VAR_1 will be ' ** MY LENGTH IS' |

CLEAR Statement

The CLEAR statement sets numeric Fields to zero and character Fields to blank.

Syntax

The CLEAR statement has the form:

CLEAR variable

where "variable" is fully or partially qualified name of either a View or a Field. If the variable being cleared has

-77-

- (1c) Either A or B or both must be a View. Nothing can ever be OVERLAYED to a Constant.
- (2) A Set is neither a constant, nor a variable, and absolutely nothing can be OVERLAYed from or to a Set.
- (3) Assume that A occurs nn times and B occurs mm times and that min denotes the smaller one of mm and nn. then

OVERLAY A to B

means exactly the same as the following Rule fragment:

DO FROM 1 TO min INDEX IX
 OVERLAY A(IX) to B(IX)
ENDO

If either A occurs multiple times and B does not or A occurs once and B multiple times, then OVERLAY A TO B is equalent to either

OVERLAY A(1) TO B

or

OVERLAY A TO B(1)

- (4) It is permitted to OVERLAY a Field with a View or vice versa, as long as the Field is of type "Character." The reason is that Views themselves re considered to be of type "Character" (COBOL convention).
- (5) It is permitted to OVERLAY a constant (literal or symbol) to a View, as long as the constant is of type "Character."

-78-

APPENDIX E: Flow of Control With a Rule

The Rules Language employs the usual flow of control constructs: IF and IF ... ELSE for conditions execution, CASEOF for selection of one of several alternatives, and DO...WHILE for control of repetitive loops.

Conditions

A condition is a mix of data items, relational operators, Boolean operators, and parentheses. A data item includes character, integer, and decimal constants, Fields, and Views. the relational and Boolean operators are:

| Operator | Meaning |
|----------|--------------------------|
| = | Equal to |
| <> | Not equal to |
| < | Less than |
| <= | Less than or equal to |
| > | Greater than |
| >= | Greater than or equal to |
| INSET | Is a member of |
| AND | Conjunction |
| OR | Inclusive disjunction |
| NOT | Negation |

Figure 1 Relational and Boolean Operators

A condition is either a single relational condition (henceforth called a "simple condition") or two or more simple conditions connected with the Boolean operators. A simple condition has the form

data item relational operator data item

Some examples are:

```

CUST_BAL_AMT < 1000
GROSS_PAY OF RTAXCMPI >= FICA_CUTOFF
CUSTOMER_ID <> '1134-4'
SSN OF RTAXCMPO = '360-32-2528'
12 INSET MONTH_OF_YEARS

```

The Rules Language enforces data-type checking; comparisons can only be made between data items of like type--numeric with numeric and character with character. All of the comparisons of magnitude are available with numeric data but only tests for equality (=) and inequality (<>) are permitted with character data.

-78-

APPENDIX E: Flow of Control With a Rule

The Rules Language employs the usual flow of control constructs: IF and IF ... ELSE for conditions execution, CASEOF for selection of one of several alternatives, and DO...WHILE for control of repetitive loops.

Conditions

A condition is a mix of data items, relational operators, Boolean operators, and parentheses. A data item includes character, integer, and decimal constants, Fields, and Views. the relational and Boolean operators are:

| Operator | Meaning |
|----------|--------------------------|
| = | Equal to |
| <> | Not equal to |
| < | Less than |
| <= | Less than or equal to |
| > | Greater than |
| >= | Greater than or equal to |
| INSET | Is a member of |
| AND | Conjunction |
| OR | Inclusive disjunction |
| NOT | Negation |

Figure 1 Relational and Boolean Operators

A condition is either a single relational condition (henceforth called a "simple condition") or two or more simple conditions connected with the Boolean operators. A simple condition has the form

data item relational operator data item

Some examples are:

```

CUST_BAL_AMT < 1000
GROSS_PAY OF RTAXCMPI >= FICA_CUTOFF
CUSTOMER_ID <> '1134-4'
SSN OF RTAXCMPO = '360-32-2528'
12 INSET MONTH_OF_YEARS

```

The Rules Language enforces data-type checking; comparisons can only be made between data items of like type--numeric with numeric and character with character. All of the comparisons of magnitude are available with numeric data but only tests for equality (=) and inequality (<>) are permitted with character data.

-79-

Each simple condition results in a value of either TRUE or FALSE. Two of these values can be combined with the Boolean operators AND and OR and the sense of one of them can be reversed with NOT.

For example,

CUSTOMER_ID <> '1134-4' AND GROSS_PAY >= FICA_LIMIT

is TRUE if both CUSTOMER_ID is not equal to '1134-4' and GROSS_PAY is greater than or equal to FICA_LIMIT; otherwise the condition is FALSE. It is assumed that CUSTOMER_ID is either CHAR, VARCHAR, or a View and GROSS_PAY AND FICA_LIMIT are INTEGER, DECIMAL, or described with numeric PIC elements. Further, for this condition to make sense, the relational operators <> and >= must be applied before the two resulting values are ANDed. This implies all the operators have hierarchy of precedence. The usual hierarchy is used, and it is given in the following table.

| Operator | Precedence |
|---------------------|------------|
| INSET | Highest |
| =, <>, <, <=, >, >= | |
| NOT | |
| AND | |
| OR | Lowest |

Figure 2 Precedence - Relational and Boolean Operators

The concept of operator precedence is well known from the everyday usage of arithmetic calculations, where multiplication and division are executed before addition and subtraction. In the same way, a test for inequality is executed before two conditions are ANDed, which itself will be done before any conditions will be ORed.

parentheses may be used to override the operator precedence. For example,

NOT X = Y or C > D is equivalent to (NOT X=Y) or (C>D),
which itself is equivalent to X <> Y or C > D.

however,

NOT (X = Y OR C > D) is equivalent to X <> Y AND C <= D (because it is true in general that NOT (X OR Y) is the same as (NOT X) AND (NOT Y)).

-81-

The following expressions are valid:

```

JAN IN MONTHSET INSET NAMESET      *>Results=FALSE<*
AL  INSET NAMESET                   *>Results=TRUE<*
AL <= JAN IN MONTHSET               *>Results=FALSE<*
AL <= JAN IN NAMESET                *>Results=TRUE<*
AL <= AL IN NAMESET                 *>Results=TRUE<*
AL <= AL NAMESET                     *>Results=FALSE<*

SHRTINT INSET NAMESET
LONGINT INSET MONTHSET

```

Of course, nothing can be said at code generation time about the results of the last two expressions because they involve variables. It might have come as a surprise to the reader that:

JAN IN NAMESET INSET MONTHSET evaluates to FALSE?

Comparison of like types is rigidly enforced. Comparisons between numeric data and non-numeric are always errors. The following table gives conditions arising between the comparison of two character types and or two numeric types. The significance of note (1), however, varies according to the types being compared.

| First | Second | | |
|------------|---------|------------|----------|
| | Literal | Set symbol | Variable |
| Literal | Error | Warning | (1) |
| Set Symbol | Warning | Warning | (1) |
| Variable | (1) | (1) | OK |

Figure 3 Character and Numeric Comparison Error Conditions

- (1) Comparing a character constant to a character variable is permitted so long as the length of the constant is less than or equal to the maximal length of the variable. Otherwise a syntax error is generated.

It is an error to compare a large (in absolute value) numeric literal with a variable which is "too small" to accommodate a value that large. For example, an SMALLINT Field can accommodate values ranging from -32,768 to +32,767. Let SMALL_INT be an SMALLINT Field. The following comparison is not permitted by the Rule code generators:

-82-

SMALL_INT > 1000000

Even more subtle errors are detected. Let POS-NUMBER be a decimal number described by the picture "PIC 9999/V99" in the HPS Repository. Then the next example condition is also not permitted:

POS_NUMBER < 0

this is an error because POS-NUMBER does not have a leading sign picture element S.

Not: On the comparison of character type items assume that X= 'ABC' and Y = 'ABC' and Z= 'ABC '. Then X and Y and Z are all equivalent as far as testing them for equality is concerned.

In other words, the complex condition X = Y and Y=Z and X=Z evaluates to TRUE. In addition, note that the ASCII collating sequence and not the EBCDIC collating sequence applies for relative comparison. Given the above definition, the condition

x <= '1BC'

yields TRUE because the ASCII value of '1' is 49, whereas the ASCII value of 'A' is 65.

IF STATEMENT

IF..., IF ... ELSE (Conditional Execution)

Syntax

The IF statement has the general form

```
      IF condition
      statement ...

      [ELSE
      statement ...]

      ENDIF
```

where "condition" is a logical expression which may be either TRUE or FALSE. "Statement" represents a Rules Language statement, and "..." means that the immediately preceding item may be repeated, and [] implies, and [] implies that what is contained within [and] is optional. A then is not allowed in the IF statement.

CASEOF Statement**CASE (Selection of One of Several Alternatives)**

The CASEOF statement selects one of several alternative execution paths on the basis of the value of a character or numeric Field. It is equivalent in meaning to several nested IF ... ELSE STATEMENTS.

Syntax

The CASEOF statement has the following form:

```
CASEOF VARIABLE
CASE literal [ literal ] ...
    [ statement ... ]
[ CASE literal [ literal ] ...
    [ statement ... ] ] ...
[ CASE OTHER
    [ statement ... ] ]
ENDCASE
```

The notation used here to express the syntax is more complex than that used for the IF statement. Square brackets ([and]) enclose items that may be omitted. Ellipses (...) indicate that the immediately preceding item may be repeated. Putting these two conventions together means that [statement ...] is equivalent to no statements, anything within [and] is optional) or one statement, two statements (...following statement means that it can be repeated), three statements, and so on.

Semantics

The CASEOF statement is a shorter way of expressing the same flow of control using nested IF ... ELSE statements. The variable is compared to the list of literals following the first CASE reserved word. If it is equal to any of them then the statements following the first CASE are executed up to but not including the second CASE; flow of control then passes to the statements following the ENDCASE.

If the variable is not equal to any of the literals following the first CASE clause, then it is compared to the literals following the second CASE. If the variable is equal to any of them, then the statements following the second CASE clause up to, but not including the third CASE are performed; flow then passes to the statements following the ENDCASE.

-82-

SMALL_INT > 1000000

Even more subtle errors are detected. Let POS-NUMBER be a decimal number described by the picture "PIC 9999/V99" in the HPS Repository. Then the next example condition is also not permitted:

POS_NUMBER < 0

this is an error because POS-NUMBER does not have a leading sign picture element S.

Not: On the comparison of character type items assume that X= 'ABC' and Y = 'ABC' and Z= 'ABC '. Then X and Y and Z are all equivalent as far as testing them for equality is concerned.

In other words, the complex condition X = Y and Y=Z and X=Z evaluates to TRUE. In addition, note that the ASCII collating sequence and not the EBCDIC collating sequence applies for relative comparison. Given the above definition, the condition

x <= '1BC'

yields TRUE because the ASCII value of '1' is 49, whereas the ASCII value of 'A' is 65.

IF STATEMENT

IF..., IF ... ELSE (Conditional Execution)

Syntax

The IF statement has the general form

```
IF condition
statement ...
```

```
[ELSE
```

```
statement ...]
```

```
ENDIF
```

where "condition" is a logical expression which may be either TRUE or FALSE. "Statement" represents a Rules Language statement, and "..." means that the immediately preceding item may be repeated, and [] implies, and [] implies that what is contained within [and] is optional. A then is not allowed in the IF statement.

-85-

```
        statement 4
        statement 5

        ELSE

        statement 6

        ENDIF
    ENDIF

ENDIF
```

Observe that the CASEOF statement implicitly uses tests for equality between a variable (Field or View) and a constant. The type checking enforced with the IF statement is also enforced here. Constants appearing in the CASE clauses must have the same type as the variable appearing on the CASEOF clause.

DO Statement

The DO ... ENDDO construct provides control of repetitive loops and there are two varieties--one with an explicit loop-counting mechanism and one without.

Syntax

The form of the DO ... ENDO loop control structure is as follows:

```
        DO

        [ statement ... ]

        WHILE condition

        [statement ... ]
```

UNDDO

The form with the explicit loop-counting mechanism is:

```
        DO [FROM integer data item]
          [ TO integer data item]
          [ BY integer data item]
          [ INDEX integer variable]

          [ statement ... ]

        [ WHILE condition ]

          [ statement ... ]

        ENDDO
```

The "condition" appearing in the WHILE clause is the same type of condition as described above in the section of the IF

-86-

statement. An "integer variable" is either a Field which is defined as INTEGER in the Repository or is a DCL local to the Rule. An "integer data item" is a literal, a Set constant, or the name of a Field defined as INTEGER. If needed to resolve any ambiguity, partial qualification and subscripting may be required with Field names.

Some examples of DO-loops follows:

```

DO
  WHILE FICA OF RTAXCMPI <FICA_MAX_IN PARAMETERS
    statement1
    statement2

...
ENDDO

DO
  statement1
  statement2

...
WHILE TOTAL_AMT > TOTAL_LIMIT
ENDDO

DO TO LOOP_END BY STEP INDEX COUNTER OF RXYZZYI
  statement1
  statement2

...
ENDDO

DO FROM LEVEL OF RSTATUSI
  statement1
  statement2
WHILE CODES (LEVEL) <> TERM_CODE IN VALID_CODES
  statement3

...
ENDDO

```

Semantics

For the first type of DO construct--without the explicit counting mechanism--the statements between DO and ENDDO are repeated in order of appearance while the condition in the WHILE clause is TRUE. When the condition becomes FALSE, control passes from the WHILE clause to the statement following the ENDDO. Note that the while clause may appear at the top of the loop, at the bottom of the loop, or anywhere in the middle.

In the second type of DO, default values are supplied for any of the missing clauses (FROM, TO, BY). If FROM is omitted, the loop count begins at 1; if TO is omitted, the count limits is set to 32,767; if BY is omitted, the loop increment is set to 1. Note that the INDEX clause need not appear. However, at least one FROM, TO, BY, or INDEX must be given for the DO to be recognized as an indexed DO. Also, IN, FROM, TO and BY values need not be integer constant; integer Fields are also permitted.

-86-

statement. An "integer variable" is either a Field which is defined as INTEGER in the Repository or is a DCL local to the Rule. An "integer data item" is a literal, a Set constant, or the name of a Field defined as INTEGER. If needed to resolve any ambiguity, partial qualification and subscripting may be required with Field names.

Some examples of DO-loops follows:

```

DO
  WHILE FICA OF RTAXCMPI <FICA_MAX_IN PARAMETERS
    statement1
    statement2
  ...
ENDDO

DO
  statement1
  statement2
  ...
  WHILE TOTAL_AMT > TOTAL_LIMIT
  ENDDO

DO TO LOOP_END BY STEP INDEX COUNTER OF RXYZZYI
  statement1
  statement2
  ...
ENDDO

DO FROM LEVEL OF RSTATUSI
  statement1
  statement2
  WHILE CODES (LEVEL) <> TERM_CODE IN VALID_CODES
  statement3
  ...
ENDDO

```

Semantics

For the first type of DO construct--without the explicit counting mechanism--the statements between DO and ENDDO are repeated in order of appearance while the condition in the WHILE clause is TRUE. When the condition becomes FALSE, control passes from the WHILE clause to the statement following the ENDDO. Note that the while clause may appear at the top of the loop, at the bottom of the loop, or anywhere in the middle.

In the second type of DO, default values are supplied for any of the missing clauses (FROM, TO, BY). If FROM is omitted, the loop count begins at 1; if TO is omitted, the count limits is set to 32,767; if BY is omitted, the loop increment is set to 1. Note that the INDEX clause need not appear. However, at least one FROM, TO, BY, or INDEX must be given for the DO to be recognized as an indexed DO. Also, IN, FROM, TO and BY values need not be integer constant; integer Fields are also permitted.

-87-

Finally, a WHILE condition may also be included with an indexed DO and it may appear anywhere within the loop.

-89-

- . the Target Environment of the USEing Rule is PC, or
 - . The Target Environment of the used Rule is PC.
- However, Rules on other machines cannot invoke a PC-based Rule (except indirectly through the ASYNC facility).

RETURN Statement

An invoked Rule program executes the RETURN statement to transfer control back to the invoking Rule. The RETURN statement may be placed anywhere in a program. The Rules Language code generator places an implicit RETURN at the end of each Rule program.

Transfer of Data Between Rules

As was mentioned in the previous section, Rules may have at most one Input View and one Output View defined in the Repository. With this in mind, the following restrictions apply concerning flow in the transfer of data:

- . A Rule may not modify its Input View unless it also happens to be its Output (i.e, INOUT).
- . A Rule may not modify the Output View of a module it calls unless it is the same as the modules Input.
- . A Rule may not share its Input View with that of its child's Output. View sharing is permitted between Rules of the same level and when the Views are the same Usage (i.e., both Input).
- . Output View of Rules and the Input Views of modules it calls, are cleared upon invocation, as are locally declared Views.

Converse Support Structures

The CONVERSE WINDOW supports communication between Rules programs in the personal computer environment and the user of a system.

The statement has the form:

CONVERSE WINDOW WINDOW_NAME

where WINDOW_NAME is the name given to the display when it was entered into the HPS Repository.

Note that there is a difference between WINDOW_NAME and Window View. Each WINDOW_NAME is associated with a View which is called its Window View. Not every View in the Repository can be used as a Window View. A Window View can only have a 01 level, 03 level, and 05 level. Note that a Rule can have more than one

-90-

Converse statement but never more than one Window Name. Note also that only PC-based Rules can have Converse statements.

Asynchronous Support Structures

This appendix summarizes the rule language constructs that invoke the unsolicited data output (UD) processes. Unsolicited data allows a one rule to send data to another Rule without the latter Rule asking for it. This is supported using the following statements:

- . ASYNC ATTACH
- . ASYNC DETACH
- . ASYNC REFRESH
- . ASYNC ROUTE

The following discussion relates to a hardware configuration with an architecture comprised of an IBM mainframe supported by the CICS operating system, a Stratus mini computer and a bank of PC work stations. All of these constructs are currently confined to PC based rules. There also exist Stratus subroutines to execute the following statements.

In defining the PC constructs, we will assume the existence of the following Rule entity instances:

- . rpc1 A PC Rule conversing Window wpc1
- . rpc2 A PC Rule conversing Window wpc2
- . rpc3 A PC Rule refreshing wpc1
- . rpc4 A PC Rule refreshing wpc2
- . rst1 A Stratus Rule invoking a component
 which uses the send_message
 subroutine.

ASYNC ATTACH STATEMENT

Syntax

The ASYNC ATTACH statement has the following form:

ASYNC ATTACH RULE rpc3 VIA RULE rst1

This command, will initiate the receiving of unsolicited input by the PC Rule rpc3 sent by the Stratus Rule rst1.

-90-

Converse statement but never more than one Window Name. Note also that only PC-based Rules can have Converse statements.

Asynchronous Support Structures

This appendix summarizes the rule language constructs that invoke the unsolicited data output (UD) processes. Unsolicited data allows a one rule to send data to another Rule without the latter Rule asking for it. This is supported using the following statements:

- . ASYNC ATTACH
- . ASYNC DETACH
- . ASYNC REFRESH
- . ASYNC ROUTE

The following discussion relates to a hardware configuration with an architecture comprised of an IBM mainframe supported by the CICS operating system, a Stratus mini computer and a bank of PC work stations. All of these constructs are currently confined to PC based rules. There also exist Stratus subroutines to execute the following statements.

In defining the PC constructs, we will assume the existence of the following Rule entity instances:

- . rpc1 A PC Rule conversing Window wpc1
- . rpc2 A PC Rule conversing Window wpc2
- . rpc3 A PC Rule refreshing wpc1
- . rpc4 A PC Rule refreshing wpc2
- . rst1 A Stratus Rule invoking a component
 which uses the send_message
 subroutine.

ASYNC ATTACH STATEMENT

Syntax

The ASYNC ATTACH statement has the following form:

ASYNC ATTACH RULE rpc3 VIA RULE rst1

This command, will initiate the receiving of unsolicited input by the PC Rule rpc3 sent by the Stratus Rule rst1.

-91-

Semantic Checking

ATTACH can only be used if:

Execution environment (Repository attribute of the
ATTACH'ed rule is PC and VIA RULE is STRAT

ASYNCH DETACH Statement**Syntax**

The ASYNCH DETACH statement has the following form:

ASYNCH DETACH RULE rpc3 VIA RULE rst1

This command performs the exact reverse of ATTACH. It may only be used after an ATTACH command, at which point it will discontinue the receiving of unsolicited input by the PC Rule rpc3 sent by the Stratus Rule rst1.

Semantic Checking

Same as for ATTACH.

ASYNCH ROUTE Statement**Syntax**

The ASYNCH ROUTE statement has the following form:

ASYNCH ROUTE RULE rpc3 VIA RULE rpc4

This command will switch the action of refreshing a Window from the PC Rule rpc3 to the PC Rule rpc4.

Semantic Checking

ROUTE can only be used if:

Execution environment (Repository attribute) of both
Rules is the PC.

ASYNCH REFRESH Statement**Syntax**

The ASYNCH REFRESH statement has the following form:

| | |
|-----------------------|-------------|
| ASYNCH REFRESH WINDOW | wpcl |
| (FIELD | field_name |
| VIEW | view_name |
| OCCUR | field_occur |
| BEEP | |
| FLASH) | |

-93-

i is either an interger or a field of type INTEGER.

Semantic Checking

- . WINDOW clause is mandatory.
- . The VIEW clause, if given, must specify a sub-View of the View identified by the WINDOW clause.
- . The FIELD clause, if given, must specify a Field name Included in the View identified by the View clause.
- . If OCCUR is used the View identified by the VIEW clause must be an occurring View.
- . The OCCUR literal must be within bounds.

Guidelines Concerning Usage of ASYNCH

- . A Rule cannot contain both CONVERSE statements and ASYNC statements
- . A Rule which CONVERSES cannot USE a Rule with ASYNC statement and vice versa.
- . A Rule given the Repository defined attribute ASYNC, cannot have a USE....NEST statement (because NEST implies that somewhere in the hierarchy of the rules it calls or in itself, there will be a Rule with a CONVERSE).

Each WINDOW_NAME is associated with a View, called Window View. Not any View in the Repository can be used as a Window View. A Window View can only have an 01 level, 03 level, and 05 level. Note that a Rule can have more than one Converse statement but never more than one Window Name. Note also that only PC-based Rules can have Converse statements.

What is claimed is:

1. A method of operating a data processor to generate a computer program, comprising the steps of:

a. inputting information relating to functions of the computer program by specifying the information according to a preselected set of objects; said objects comprising a preselected set of entities and relationships;

b. linking each entity to one or more other entities by the pre-selected set of relationships;

c. storing the linked set of entities and relationships in a storage area;

d. inputting logic constructs referencing objects stored in the storage area;

e. relating the logic constructs to pre-selected ones of the stored entities; and

What is claimed is:

1. A method of operating a data processor to generate a computer program, comprising the steps of:
 - a. inputting information relating to functions of the computer program by specifying the information according to a preselected set of objects; said objects comprising a preselected set of entities and relationships;
 - b. linking each entity to one or more other entities by the pre-selected set of relationships;
 - c. storing the linked set of entities and relationships in a storage area;
 - d. inputting logic constructs referencing objects stored in the storage area;
 - e. relating the logic constructs to pre-selected ones of the stored entities; and

-95-

- f. storing the logic constructs and information on the related preselected ones of the entities in the storage area.
- 2. The method of Claim 1 including the further step of utilizing the logic constructs and the referenced objects to generate corresponding source code.
- 3. The method of Claim 2, comprising the further step of storing the source code in the storage area.
- 4. The method of claim 2 comprising the further step of utilizing the corresponding source code to generate executable computer code.
- 5. The method of claim 4 including the further step of storing the executable computer code in the storage area.
- 6. The method of Claim 1 comprising the further step of
 - a. inputting additional information relating to the function of a second computer program;
 - b. searching the storage area for objects matching the additional information; and

-97-

c. storing the linked set of new entities and relationships in the storage area.

11. The method of Claim 10 including the further steps of:

- a. providing a plurality of hardware environments; coupled to one another to form an interconnected computer network; and
- b. specifying for each logical construct one of the plurality of hardware environments for execution.

12. The method of Claim 11 comprising the further steps of:

- a. storing in the storage area information regarding each one of the plurality of hardware environments; and
- b. utilizing each logical construct and information regarding the respective specified hardware environment for the logical construct to generate source code adopted to be executable in the specified execution environment.

13. The method of Claim 12 wherein steps a-f of Claim 1 are carried out in a pre-selected one of said hardware environments.

14. The method of Claim 13 comprising further steps of:
 - a. transmitting the source code generated in respect of each of the logical constructs to the respective specified hardware environment; and
 - b. utilizing the source code to generate executable computer code at the respective hardware environment.
15. The method of Claim 14 wherein the storage area comprises a Central Repository located at a preselected one of the hardware environments.
16. A method of operating a data processor to generate a computer program, comprising the steps of:
 - a. inputting information descriptive of a real-world circumstance;
 - b. using the information to build a model comprising a pre-selected set of entities and relationships reflective of the information;
 - c. using the information to build a further model comprising a pre-selected set of entities and relationships that describe the function of the computer program in respect of the real-world circumstance;

14. The method of Claim 13 comprising further steps of:

- a. transmitting the source code generated in respect of each of the logical constructs to the respective specified hardware environment; and
- b. utilizing the source code to generate executable computer code at the respective hardware environment.

15. The method of Claim 14 wherein the storage area comprises a Central Repository located at a preselected one of the hardware environments.

16. A method of operating a data processor to generate a computer program, comprising the steps of:

- a. inputting information descriptive of a real-world circumstance;
- b. using the information to build a model comprising a pre-selected set of entities and relationships reflective of the information;
- c. using the information to build a further model comprising a pre-selected set of entities and relationships that describe the function of the computer program in respect of the real-world circumstance;

-99-

- d. providing certain ones of the entities as rule entities; each one of the rule entities describing an aspect of the function of the computer program and its relationship to other rule entities; and
- e. providing a rules language comprising logic statements which provide executable functionality of the aspects described in the rules entities.